A Study of Rootkit Stealth Techniques and Associated Detection Methods

THESIS

Daniel D. Nerenberg, 1st Lieutenant, USAF

AFIT/GCE/ENG/07-10

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/07-10

A Study of Rootkit Stealth Techniques and Associated Detection Methods

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Daniel D. Nerenberg, B.S.C.E.

1st Lieutenant, USAF

March 2007

AFIT/GCE/ENG/07-10

A Study of Rootkit Stealth Techniques and Associated Detection Methods

Daniel D. Nerenberg, B.S.C.E.
1st Lieutenant, USAF

Approved:

/signed/	5 Mar 2007
Major P. Williams, PhD (Chairman)	date
/signed/	5 Mar 2007
Dr. R. Baldwin (Member)	date
/signed/	5 Mar 2007
Dr. R. Raines (Member)	date

Abstract

In today's world of advanced computing power at the fingertips of any user, computer security should be a primary concern. Information is power and this power is within the computer system. If the information within computer systems cannot be trusted then the power that comes from such information cannot be properly used. Rootkits are software programs that are designed to establish and maintain an environment in which malware may hide on a computer system after successful compromise of that computer system. Rootkits cut at the very foundation of the trust in information and subsequent power.

This thesis examines rootkit hiding techniques, rootkit finding techniques and develops attack trees and defense trees to identify deficiencies in detection and further increase the trust in information systems. The developed attack and defense trees identified that enumeration is not sufficient to defend against rootkits. A developed classification of rootkits helps fill the gaps in enumeration of rootkit techniques and gives direction for further detection development. By fully understanding what areas need to be addressed in detection, better and more complete tools will come to fruition.

Acknowledgements

First and foremost I would like to thank my Heavenly Father for blessing me with the ability to complete this thesis. Secondly, I would like to thank my wife and children for supporting and encouraging me throughout this educational adventure. I would also like to express my sincere appreciation to my faculty advisor, Major Paul Williams, and committee members, Dr. Rusty Baldwin and Dr. Richard Raines, for their guidance and support throughout the course of this thesis effort. Their insight and experience was certainly appreciated. I would, also, like to thank Mr. Lacey for his expertise and support in everything. Special thanks to Greg Hoglund, James Butler and all those who participate in www.rootkit.com, and in the writing of *Rootkits: Subverting the Windows Kernel*; these were priceless in my rootkit research.

Daniel D. Nerenberg

Table of Contents

Page

List of Figures

Figure Page

List of Tables

List of Abbreviations

Abbreviation		Page
IDS	Intrusion Detection Systems	1
IPS	Instrusion Prevention Systems	1
RHTs	Rootkit Hiding Techniques	2
RDTs	Rootkit Detection Techniques	2
SUT	System Under Test	3
LKM	Loadable Kernel Module	19
SSDT	System Service Descriptor Table	24
IDT	Interrupt Descriptor Table	24
IRP	I/O Request Packet	24
IAT	Import Address Table	28
DSM	Data Structure Manipulation	30
DKOM	Direct Kernel Object Manipulation	30
FLINK	forward link	31
BLINK	backward link	31
KPCB	Kernel Process Control Block	36

A Study of Rootkit Stealth Techniques and Associated Detection Methods

I. Introduction

1.1 Motivation

Computer networks and their associated devices have been under attack by hackers for some time, and defenses have been developed to protect against a myriad of attacks. However, in recent years a powerful new class of software called rootkits and also known as stealth technology has been developed.

Rootkits are software programs that establish and maintain an environment in which malware can hide on a computer system after successful compromise of that computer system. Since the inception of rootkits in the 1980s [8], rootkits have improved in both their techniques and ability to protect hackers and their tools from being discovered. McAfee(R) Avert(R) Labs indicates that over the last three years the "incident rate of stealth technology has increased by more than 600 percent [34, 35]", and "The number of rootkits submitted ... in 2006 compared to the first quarter of 2005 increased by nearly 700%. The number of Windows-based stealth components dominate the landscape, with an increase of 2300% from 2001 to 2005 [34, 35]."

There is often ample time between the discovery of vulnerabilities and patching of that vulnerability to install a rootkit. Installation of a rootkit compounds the effects of the original compromise because a rootkit creates and maintains an environment in which a software entity may hide itself and its effects thus making the original compromise much more difficult to detect and remove.

There are many ways to mitigate threats to a computer system such as firewalls, Intrusion Detection Systems (IDS) and Instrusion Prevention Systems (IPS). However, because of the time gap between vulnerabilities and patches, the increasing capabilities of exploits, and the large increase in numbers of rootkits computers are

still very vulnerable. Thus, it is vital to understand how rootkits hide and how they are found in order to build a rootkit attack tree that can identify holes in detection techniques.

1.2 Overview

Rootkits create and maintain an environment for attack tools, such that a user does not know of their presence on a compromised machine. However, the rootkit does not gain access to the machine, rather, it maintains the access. In this research Rootkit Hiding Techniques (RHTs)are explored as well as the Rootkit Detection Techniques (RDTs) that are being used by some currently available tools. We analyze these hiding and discovery techniques to identify deficiencies in detection.

Attack trees for rootkits are developed which give a high level overview of important areas to defend in computer systems. An experiment in detection is conducted using a rootkit to find rootkits. Although operating systems other than Windows(R) are discussed, this research focuses on Windows(R) rootkits for two reasons: First, operating systems differ in names for various tables, in functionality, and in stability but the concepts are roughly the same (i.e., each has hardware interface(s), scheduler, kernel, libraries, system calls, and distinctions between user and kernel level permissions) meaning the same general conclusions and attack trees can be shared with minor modifications to suit each system (cf., Appendicies A and B.) Second, according to McAffee(R) "The share of Linux-based techniques has gone from a high of roughly 72 percent of all malware stealth components in 2001 to a negligible number in 2005, while the number of Windows(R)-based stealth components has increased by 2,300 percent in the same time period [35]."

1.3 Research Statement

In this research we study the methods by which rootkits hide and also how they can be detected. These hiding and finding techniques are used to create attack trees from which deficiencies in current detection techniques can be identified, identified

deficiencies can be used to increase the effectiveness of computer system defenses. This effort includes examples of current rootkits and rootkit detectors.

1.4 Thesis Organization

This chapter gives motivation, an overview, and a research statement as well as the organization of the thesis. Chapter Two contains a literature review that encompasses operating systems, classification systems, trusted computing, exploits versus patches, and gives a general background on rootkits. Chapter Three describes various hiding techniques as well as examples of these techniques. The hiding techniques are followed by various finding techniques used by RDTs. Chapter Four explains the research findings and describes the System Under Test (SUT) as well as experimental results. Finally, Chapter Five presents conclusions and recommendations for future research.

II. Literature Review

2.1 Background

Attacks against computer systems have a wide range of effects and purpose. Some attacks are meant to be malicious in various degrees, while others are simply for curiosity. Regardless of the intent of an attack, we require the ability to defend our computer systems and allow/deny access to our computer systems as we see fit, not as an intruder may see fit. In attempt to protect and study the defense of our computer systems many classification systems or taxonomies have been developed and are discussed in Section 2.3.

Computer attacks may have a long period of time in which to occur because of the length of time between discovery of a vulnerability (a software flaw that allows other than intended operations to occur) and the application of a patch (a piece of code used to replace/fix software vulnerabilities), which is further explained in Section 2.5 of this thesis. If vulnerabilities are left unchecked and a threat (something willing to take advantage of the vulnerability) exists then our computer systems are at risk. This risk ranges from exposure (the simple loss of data) to the complete loss of control over a machine. The next step, once a successful computer attack has occurred, is that of maintaining control; this control can be maintained by rootkit software. Just as a country does not or should not fight a battle and ignore the war, a computer attack is simply the beginning (a battle) to lay the ground work for maintaining control over the computer asset (the war). As noted by Anton Chuvakin of iDEFENSE Labs, "Rootkits are automated software packages to setup and maintain an environment on a compromised machine." According to Chuvakin, the first evidence in the public domain of Rootkits was discovered in the mid 1990s [15]. Rootkits are explained further in Section 2.6, of this thesis. This thesis will seek to help identify rootkit stealth techniques and identify deficiencies in rootkit detection techniques. By identifying deficiencies in detection we can learn to better detect and subsequently remove rootkits. The subsequent sections of this chapter will give the

reader a background in operating systems, classification systems, trusted computing, exploits vs patches, and rootkits.

2.2 Operating Systems

In order to understand how a rootkit might install itself and hide in a computer we must understand a computer from the architecture level. Silberschatz, Gagne, and Galvin in their book *Operating System Concepts*, divide a computer system into four main areas: hardware, the operating system, application software, and users using Figure 2.1 to illustrate [59].

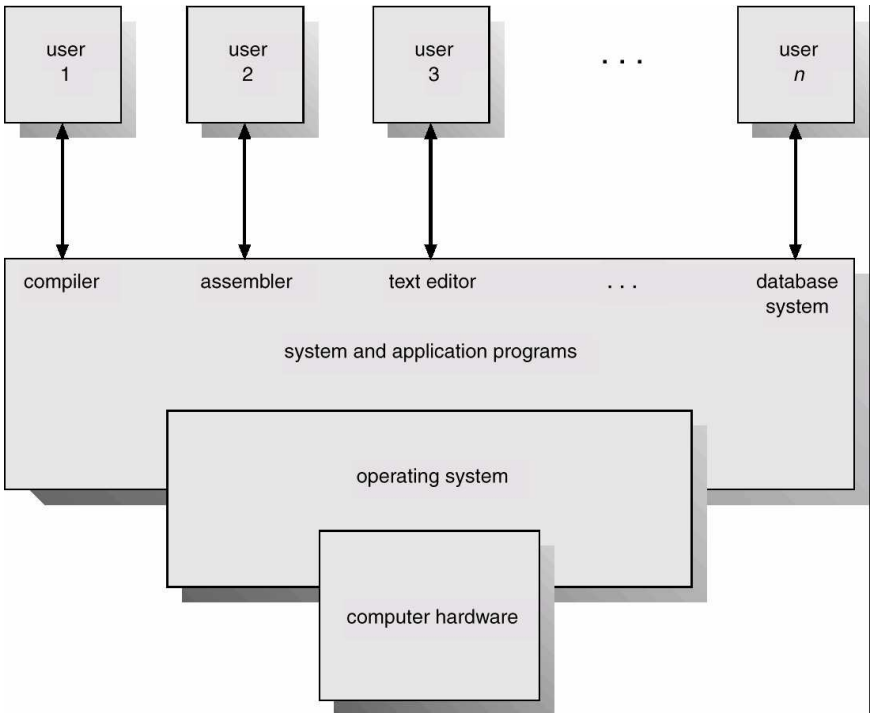


Figure 2.1: Abstract view of the components of a computer system [59].

The user of a computer system uses hardware input/output devices such as the keyboard and mouse in order to start desired tasks. Those hardware I/O devices interact with application software in order to accomplish the desired tasks. The operating system is the intermediary between the hardware, user, and application software which acts as a manager between the user, the hardware, and the application

software of a computer system [59, 65]. A good example used by Silberschatz et al, is that the operating system is "Like a government, it performs no useful function by itself. It simply provides an environment within which other programs can do useful work" [59].

If we look at Figure 2.1, we notice that there are a few layers within the diagram in which a rootkit could insert itself, namely: at the application level, the operating system level, and at the hardware level. Although inserting rootkits at the hardware level may be feasible, as documented by John Heasman in his paper *Implementing and Detecting a PCI Rootkit* [28], for this thesis, we will focus on the first two levels, paying special attention to the operating system level with its many sub-levels. Within each operating system is a structure called a kernel which, according to McKusick and Neville-Neil in their book *The Design and Implementation of the FreeBSD Operating System*, "...is a small nucleus of software that provides only the minimal facilities necessary for implementing additional operating-system services" [43]. The size and functionality of kernels have increased over the years but the intent remains the same, such that the kernel controls what and when each thread, process, or task is run and for how long. The following subsections will give the background needed for the Windows, Linux, and BSD OSs. Furthermore, these sections will show what each of the main divisions are in each OS. However, as will be seen, the operating systems are very similar at least from a high level component view and attack perspective.

Not all of the tables or important structures will be named equivalently between operating systems but the underlying architecture of how an operating system behaves when an input/interrupt occurs is similar. Max Bruning states in his article *A comparison of Solaris, Linux, and FreeBSD* that, "Once you get past the different naming conventions, each OS takes fairly similar paths toward implementing the different concepts" [6]. For example, in Linux or Windows, when an interrupt is triggered, the interrupt handler takes over. The interrupt handler will then check an interrupt descriptor table (IDT) to know how to handle the particular kind of interrupt/system call request, the system call table is checked and then control jumps

to the address found. Control of this process can be usurped either by changing the interrupt handler so that it uses a completely new IDT, or by changing addresses in the IDT [64]. Execution of a system call in a UNIX-like operating systems (which parallels that of a Windows OS) is shown in Figure 2.2.

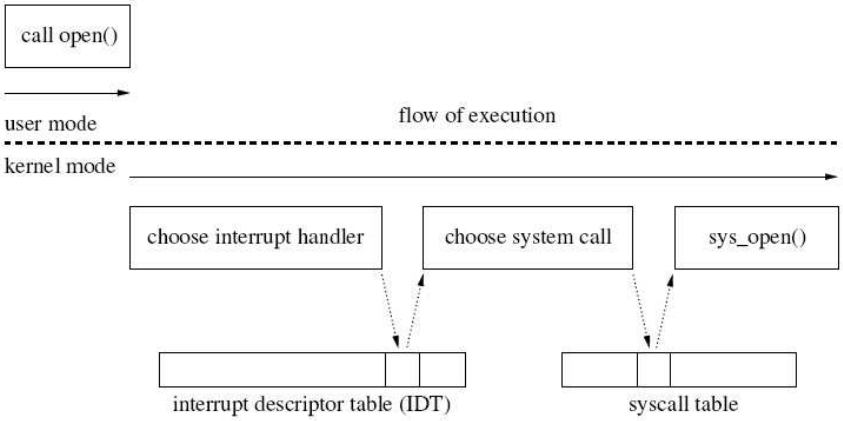


Figure 2.2: UNIX-like OS: System call flow [7].

2.2.1 Windows OS. According to Silberschatz et al, Windows was designed for "...security, reliability, Windows and POSIX application compatibility, high performance, extensibility, portability, and international support" [60]. As can be seen in Figure 2.3, the Windows OS was created in many modules. The main areas of which to take note are the Hardware-Abstraction Layer, the Kernel, the executive, and the usermode subsystems. The Kernel is the part of the operating system that decides what, when, and for how long each thread is going to run based on parameters that it receives as to priority [60].

In the Windows OS we find rootkits of three main types, namely patching (replacement of code sections), hooking (altering execution paths), and data structure manipulation (altering data structures). Windows much like any operating system is subject to attacks via patching because of the need to allow programs to run with varying permissions which can be granted by an unknowing user. However, steps have been taken to deter this concept of patching by concepts such as digital signing which can verify the integrity of applications.

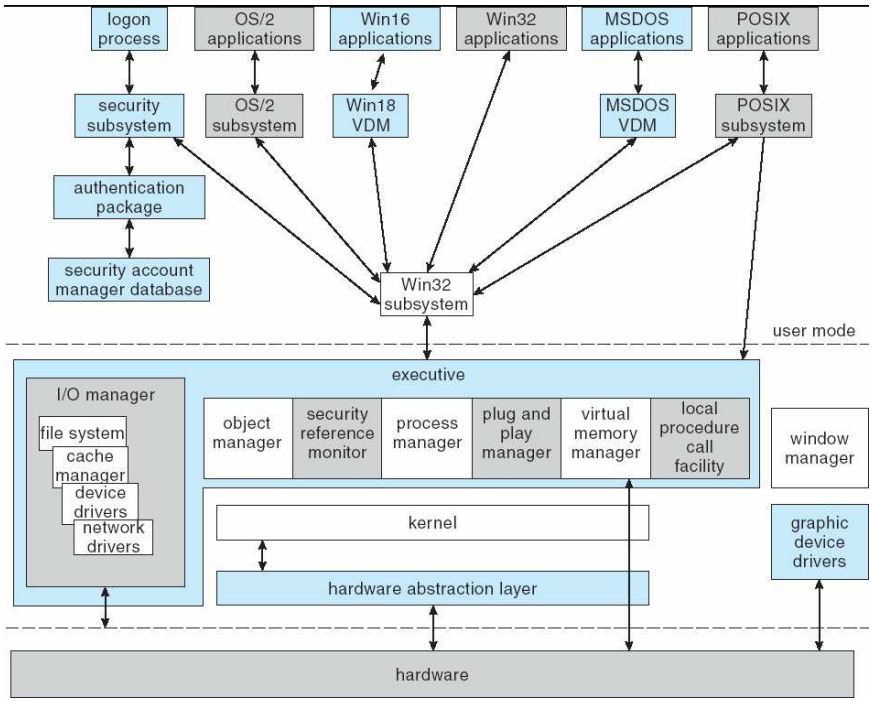


Figure 2.3: Block Diagram of Windows XP OS [59].

Windows is also subject to hooking, in part because this functionality is allowed (given the appropriate permissions) in order to monitor other processes. Finally, Windows is also subject to data structure manipulation because, as with other operating systems, there are data structures in software which can be modified. These problems are not easily addressed because if we completely removed the ability to alter applications, execution paths and structures, we would remove desired functionality and upgrade ability.

2.2.2 Linux & FreeBSD OSs. Although there are differences in implementation and some functionality between Linux and FreeBSD, their basic architecture is virtually the same and grows closer together each time there is a development in one or the other, due to cross pollination of ideas and sharing. According to Silberschatz et al, Linux was designed for "...speed, efficiency, and standardization" [59]. According to McKusick et al, "the FreeBSD kernel provides four basic facilities: processes, a filesystem, communications, and system startup" [43]. As can be seen in Figure 2.4,

the Linux OS, much like Windows, was created in many modules. The similarities between Windows and Linux can been seen such that both systems have a kernel which is surrounded by supporting modules to interact with higher level users and lower level hardware. However, a difference between Windows and Linux is in regard to processes and threads. Windows uses threads as its fundamental unit of execution while Linux does not really distinguish between processes and threads, rather, Linux generally refers to both as tasks [59].

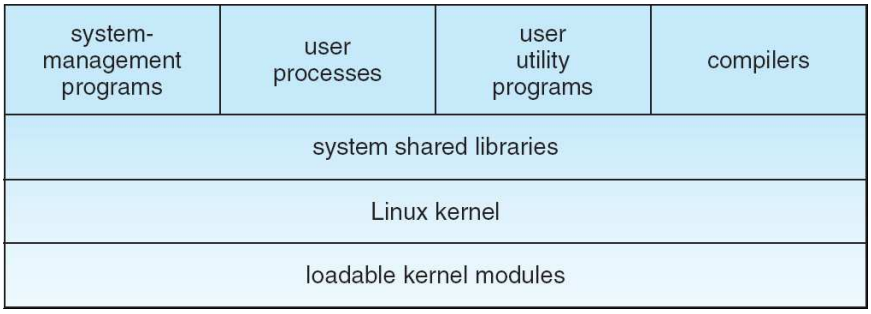


Figure 2.4: Block Diagram of Linux OS [59].

One of the first techniques used by Linux rootkits was patching which is the replacement or modification of data, files, binaries, etc. An example of such is the rootkit T0rn which "replaces *login*, *ifconfig*, *ps*, *du*, *ls*, *netstat*, *in.fingered*, *find* and *top*" [7] with modified versions of the same files. These modified versions then work to hide information from the user. For example, the *ls* command shows a list of what files are present in the current directory. The modified *ls* command would filter from view any files specified by the rootkit. One of the most common techniques for insertion of a rootkit in Linux and BSD is through a kernel loadable module which uses all three rootkit techniques. A loadable kernel module allows any information processed by the system to be modified [7]. This runtime insertion of malicious code into the kernel can be deterred by turning off the ability to load kernel modules. However, as discussed previously this can remove functionality that was used for legitimate purposes.

2.2.3 Prevention. Certainly there are configurations that make each of the various operating systems more secure such as restricting the permissions a user has

on their account (ie, user account instead of administrator or root). In the various UNIX variants, removing the ability to load kernel modules by turning off the functionality slows down the main class of rootkits because, as discussed previously, the ability to load malicious code would then need to occur in a different way such as loading through */dev/kmem*. The device files */dev/mem* and */dev/kmem* "allow the root user to arbitrarily access the contents of physical memory and kernel memory, respectively [21]." Allowing a root user to access physical and kernel memory gives all the power desired to rootkit authors that have obtained root permissions through some vulnerability.

Mandatory access controls ("When a system mechanism controls access to an object and an individual user cannot alter that access" [5]) can also be added in some UNIX variants such as Trusted Debian (aka Adamantix) [1], SE Linux [46], and Trusted BSD [68]. Other functions also exist to help secure a system, FreeBSD for example, also has a function called "jail" which restricts the user of that jail to its own associated "processes, files, and network [43]." There are also many software and hardware suites that help keep our computer systems relatively clean. Firewalls, Host Intrusion Detection Systems (HIDS), Network Intrusion Detection Systems (NIDS), Anti-Virus(AV), and various Rootkit Detectors. Another example includes the file flags and four security levels in BSD in order to further secure the kernel. These security levels are:

> "-1. Permanently insecure mode: Kernel runs at secure level 0 at all times.
>
> 0. Insecure mode: File flags may be set and reset and devices may be read from or written to according to their permissions
>
> 1. Secure mode: Superuser-setable file flags cannot be turned off. Device files for system memory */dev/mem*, */dev/kmem* and for mounted filesystems are read-only.
>
> 2. Highly secure mode: Device files for filesystems are always read-only, whether they are mounted or not. Firewall rules may not be changed" [70].

2.3 Classifications

Many classifications have been created in order to understand and protect our networks. Some of the classifications that have been used include: *achieved access*, *automation*, *attack type*, and *attack tree*.

2.3.1 Achieved Access. In Table 2.1, we see Incident Categories from Air Force Instruction 33-138, which shows the categories in which incidents fall based on the level of achieved access by the attacker. Although this type of classification system is useful in categorizing *what* has occurred in order to react it does not cover *how* it occurred in order to defend.

2.3.2 Automation. Further classifications can be found, such as classifying by degree of automation used in the attack: Manual, Semi-Automatic, and Automatic. Manual attacks are as the name implies, where the human attacker scans, breaks into, installs, and then uses some attack tool. Semi-automatic attacks, use scripts to do the scanning, breaking, and installation, which is then followed by the manual instructions to use the installed tools. Attacks are classified as automatic when the human-in-the-loop simply starts the attack script and the scanning, breaking, installation, and exploitation all occur without intervention [37].

2.3.3 Attack Type. Attack types are simply descriptive names given to attacks such as IP Spoofing, Source Routing, Routing Table Corruption, Denial Of Service Attacks (DOS), Smurf Attacks, Land Attack, Xmas Tree Attack, Teardrop, TCP/UDP Diag Services Attacks, Ping of Death, SYN Flood, and Session Hijacking [42].

2.3.4 Attack Tree. An attack tree is simply the graphical representation of how an attack reaches its goal. Bruce Schneier, the CTO of Counterpane Internet Security, describes attack trees as follows: "Attack trees provide a formal, methodical way of describing the security of systems, based on varying attacks. Basically, you

Table 2.1: Incident Categories (AFI 33-138) [2]

Category	Description
I	Root-Level Intrusion: An unauthorized person gained root-level access/privileges on an Air Force computer/information system/network device.
II	User-Level Intrusion: An unauthorized person gained user-level privileges on an Air Force computer/information system/network device.
III	Attempted Access: An unauthorized person specifically targeted a service/vulnerability on an Air Force computer/information system/network device in an attempt to gain unauthorized or increased access/privileges, but was denied access.
IV	Denial of Service (DoS)): Use of an Air Force computer/information system/network was denied due to an overwhelming volume of unauthorized network traffic.
V	Poor Security Practice: An Air Force computer/information system/network was incorrectly configured or a user did not follow established policy.
VI	Scan/Probe: Open ports on an Air Force computer/information system/network device were scanned with no DoS or mission impact.
VII	Malicious Logic: Hostile code successfully infected an Air Force computer/information system/network device. Unless otherwise directed, only those computers that were infected will be reported as a Category VII incident.

represent attacks against a system in a tree structure, with the goal as the root node and different ways of achieving that goal as leaf nodes" [57]. If we can understand how an attack reaches its goal then it will be easier to stop the achievement of such a goal. A graphical example of an attack tree for a house burglary is depicted in Figure 2.5 from the Electricity Sector Information Sharing and Analysis Center (ESISAC) [22].

The reader may note, in Figure 2.5, that "OR" gates are used to show that only one of the options is needed in order to progress through the attack path. If everyone robbing a home is: first, entering the town, second, driving down the street, third, entering the yard, and finally, going through the front door *or* the side window, then

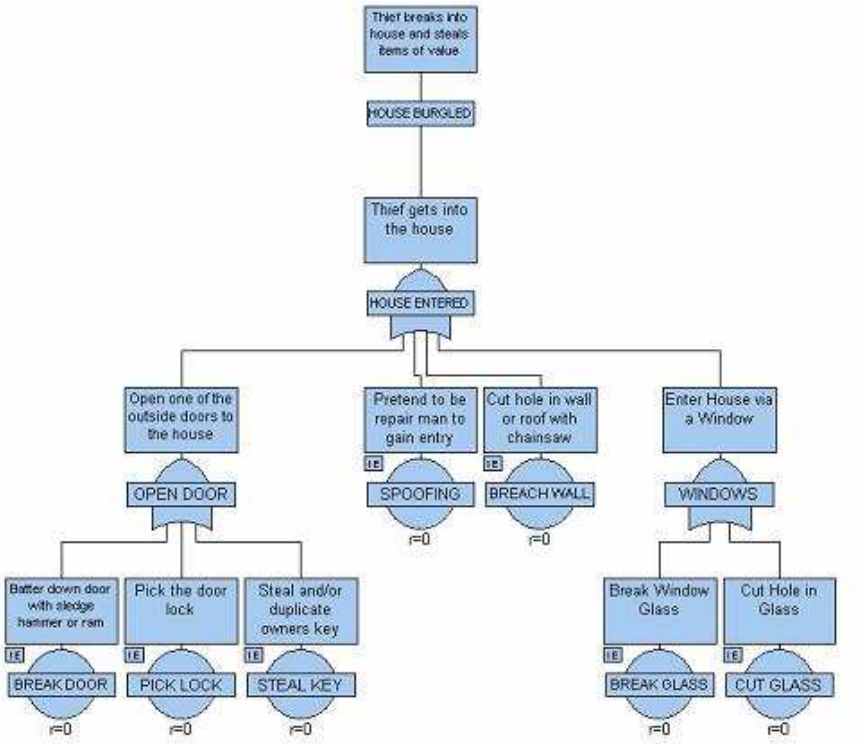


Figure 2.5: Example Threat Tree, House Burglary [22]

we focus on the door *and* the window. Although, no one wants a burgler in town, a more immediate concern is whether or not the burgler can gain access to the home.

Using an attack tree methodology, instead of only developing a new virus signature or finding a new way to block each individual new threat, we can identify the points of ingress in order to protect them better. Concentrating on points of ingress helps us to get more protection from smaller changes as well as better defense in depth. Attack trees allow us to more easily explore options for stopping groups (classes of attacks) in order to make it more difficult for the attacker to achieve their goal. For example, if we can change the "OR" gate at any of the entry points to an "AND" gate we can force the attacker to spend more time, money, and/or resources to achieve the goal.

Furthermore, if our goal as defenders is not only to slow down the attackers but to completely defend our computer resources then understanding and implementing

all of these classification techniques together will further our cause and bring us closer to our goal. For example, continuing with the previous analogy of the burglar, if we are able to track the burglar's progress (*achieved access*) then we can more readily focus our efforts on what is needed to prevent further access. If we know the level of *automation* that is being used by the burglar then we may be able to "out think" him by doing something that stops automation (perhaps encryption). If we are able to enumerate the *attack types* and separate them into those that can hurt us (unknown defense) and those that don't (known defense), then we can focus our efforts on the unknown. Although as quoted by Sun-Tzu, a Chinese general and military strategist in approximately 400 BC, "Keep your friends close, and your enemies closer", we should not forget about the attacks for which we have known defenses. By applying classification systems we can better understand where to protect our computer systems.

2.4 Trusted Computing

In an article published by the Electronic Frontier Foundation in October of 2003, Seth Schoen reports "trusted computing initiatives propose to solve some of today's security problems through hardware changes to the personal computer" [58]. According to Schoen, trusted computing architectures are being created via Microsoft Palladium (Next Generation Secure Computing Base), the Trusted Computing Group (TCG), Intel Lagrande, and AMD Secure Execution Mode (SEM). Some of the ideas within this new architectural design include memory curtaining, secure input and output, sealed storage, and remote attestation. Memory curtaining is using hardware to stop programs from reading or writing to any memory space other than their own. Secure I/O provides a "secure channel" from the I/O device to the application using the device such that the application will be able to trust its input rather than worrying about whether or not it has been altered by malicious software such as rootkits. Sealed storage uses hardware to generate machine specific keys such that data that resides on your machine can only be decrypted by your machine and the proper application.

Remote attestation is creating and sharing a cryptographic hash of your operating system and any software running on it such that if any changes are made the remote computers (computers with which you are sharing information) will be able to see that changes have occurred and not send information until the situation has been rectified [58].

The first product, which originated from IBM, of the Trusted Computing Group is the Trusted Platform Module (TPM). The TPM is a hardware component that holds "keys used to encrypt data for storage or transmission" [38]. By keeping the keys and processing of such in a stand alone unit, the ability to sniff or otherwise guess the decryption keys is dramatically reduced. Research is constantly underway in the trusted computing area. Currently, the trusted computing group recommends the following five steps [38] for implementation of trusted computing:

1. Authentication - "the binding of an identity to a subject" [5]
2. Data protection - prevention of unauthorized modification of data, possibly through encryption or other means
3. Network attestation and platform measurement - the implementation of authentication and data protection accross a network. For example, authenticating the permissions of the computer to be granted access as well as the patch levels, firewall setting [38].
4. Application protection - protection of applications from malicious activity by such things as partitioning memory so the application can not run out of it's own space
5. Content protection - digital rights management to grant usage of software only to authorized persons [38]

Efforts such as these will help improve our computer system defenses against rootkits by addressing some of the behaviors of rootkits. For example, rootkits routinely access information "outside" of their own space such as causing an application

to jump to rootkit code which is addressed by application protection. Data protection addresses an entire class of rootkits, that of patching. Network attestation will also give some protection against rootkits by helping verify those connecting to our computers.

2.5 Exploits vs Patches

As can be seen in Figure 2.6, the path from vulnerabilities being discovered, to advisories being released (which are followed by patches), is not long but it is long enough to leave time for the attacker to further compromise a computer system or to hide their tracks. In July of 2004, Deborah Radcliff of Security Focus reported, "The time between vulnerability discovery and exploit, has compressed 90 percent during the past three years the average being 11 days between discovery and exploit (well below the 23 days most enterprises need to patch)..." [49].

Patching is a useful and important step to protecting our information systems however, it is not a panacea. Scott Berinato of CSO online has an interesting view, "Slammer was unstoppable. Which points to a bigger issue: Patching no longer works. Partly, it's a volume problem. There are simply too many vulnerabilities requiring too many combinations of patches coming too fast" [4]. The number of vulnerabilities continues to increase each year. In 2005, according to cert.org [14], there were 5,990 vulnerabilities reported, which is more than 5 times as many as the year 2000, and over 35 times more than 1995. While in the first quarter of 2006 alone, there were 1597 vulnerabilities reported [14]. These figures do not account for the probable myriad of vulnerabilities found that have not been reported which make the total number of vulnerabilities that exist, increase.

The cost of patching can be estimated using the following formula and example as given by Pete Lindstrom in Information Security Magazine, "(Hours x Rate x Systems) + (Patch Failure x (Hours x Rate x Systems)) = Cost to Patch" [39].

As an example Mr. Lindstrom offers the following: "if it takes an army of $70/hour technicians one hour to patch a system, and there are 2,000 systems, the cost is $140,000. If you estimate that 5 percent of the patches fail, and figure an average of two hours of recovery time (which includes help desk and IT support activities), that's 100 systems at $140 each – another $14,000" [39]. This comes to a total of $154,000 for a single patch for one company that only has 2000 systems. This can be reduced by using automated patching but shows that patching is expensive. If the amount of patching can be reduced then the total cost of defense is thus reduced as well.

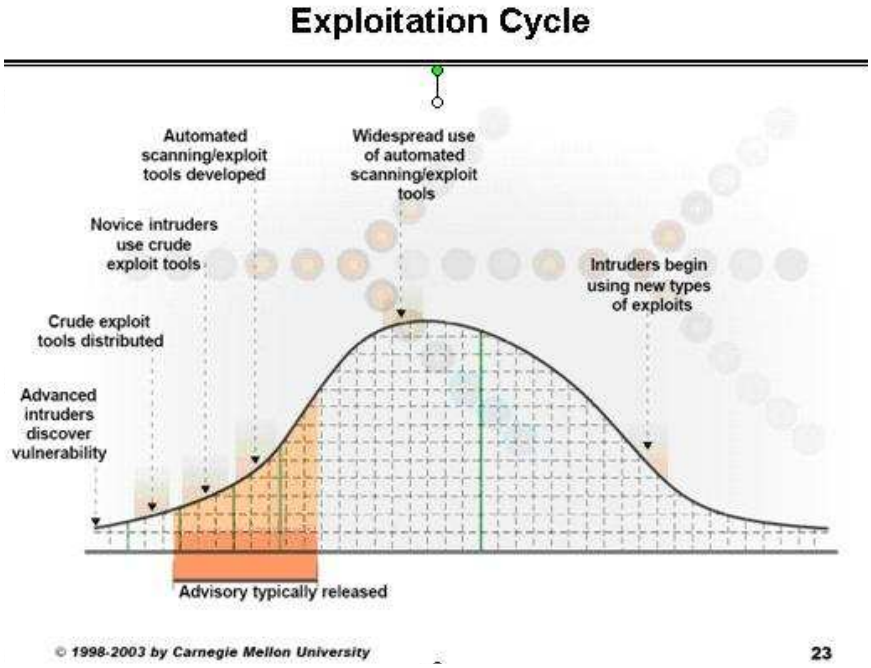


Figure 2.6: Exploitation Cycle [44]

2.6 Rootkits

Rootkits, as defined by Chuvakin, are programs used to set up and maintain an environment [15]. We add to this definition to include, protection and obfuscation of attack tools which exist on a compromised computer. According to Chuvakin, there are three main types of rootkits: Binary, kernel and library kits [15].

However, there are many ways to classify rootkits such as persistant vs memory-based. Persistant rootkits are rootkits that survive on the system (running) past a reboot whereas memory-based only rootkits live in memory and thus are effectively destroyed if a reboot occurs [11]. Joanna Rutkowska in her article, *Introducing Stealth Malware Taxonomy*, divides the classification into four types ranging from type 0 to type 3 [54].

1. Type 0 is described as any malware that uses only documented programming methods and does not directly interact with the operating system, kernel, or other any other processes.
2. Type 1 malware "modifies those resources which were designed to be constant, like e.g. in-memory code sections of the running kernel and/or processes."
3. Type 2 malware then is the counterpart to type 1 which expects that malware will modify resources that are not constant such as data sections (changing pointers in a kernel data structure is cited as an example of type 2).
4. Type 3 malware uses hardware virtualization and exists outside of the system structures. Rutkowska also created an example of type 3 as a proof of concept called Blue Pill.

In the following subsections we will briefly explain some rootkit types in order to give a background on rootkits. This classification system by type is very useful, however, for this thesis and in subsequent chapters we will divide mainly into three categories inspired by Hoglund et al [8]: Hooking, Patching, and Data Structure Manipulation (DSM). Each of these categories can occur at two levels, user and kernel [8].

2.6.1 Binary Rootkits. Binary rootkits replace common binary files with modified binary files so that when a call to that binary occurs it does as the attacker wishes rather than as originally designed [15]. An example of such, is the Linux

rootkit T0rn which "replaces *login*, *ifconfig*, *ps*, *du*, *ls*, *netstat*, *in.fingered*, *find* and *top* by manipulated versions" [7].

2.6.2 Kernel Rootkits. Kernel rootkits are probably the most dangerous kind of rootkit in that they have complete control over the machine all the way to the root level. Kernel rootkits use kernel calls to gain access to hardware rather than simply system calls which then use kernel calls. These modified kernel calls then allow the attacker to control what the computer reports to its other "trusted" applications and resources. One example of how a rootkit installs is via a Loadable Kernel Module (LKM). An LKM is used in normal functionality of a system to install various applications and their associated drivers and libraries. However, rootkits also take advantage of this functionality to insert themselves directly into the kernel. A computer has various levels of "trust" which are referred to as privileges.

"A privilege in a computer system is a permission to perform an action. Examples of various privileges include the ability to create a file in a directory, or to read or delete a file, access a device, or have read or write permission to a socket for communicating over the Internet" [66]. When working with the Intel x86 architecture we refer to these privileges as rings. As seen in Figure 2.7, Ring "zero" is where the kernel operates and has full and unrestricted control. Ring "three" is where the user has control but must use system calls in order to request information from hardware at ring zero. Ring "three" is also referred to as "userland" [8].

Of note is that most OSs only use ring zero and ring three thus separating users into somewhat restricted or not restricted at all, when it could be very useful and important to have multiple levels of permission. For example, in our banks we don't have only a single differentiation between access modes (access to all or access to some). Rather, we have access to some (e.g., user), access to more than some (e.g., teller), access to most (e.g., manager), access to all (e.g., owner with administrator and manager). If we used all four levels or rings it would add more levels of difficulty for attackers thus causing them again to use more resources in order to succeed. If

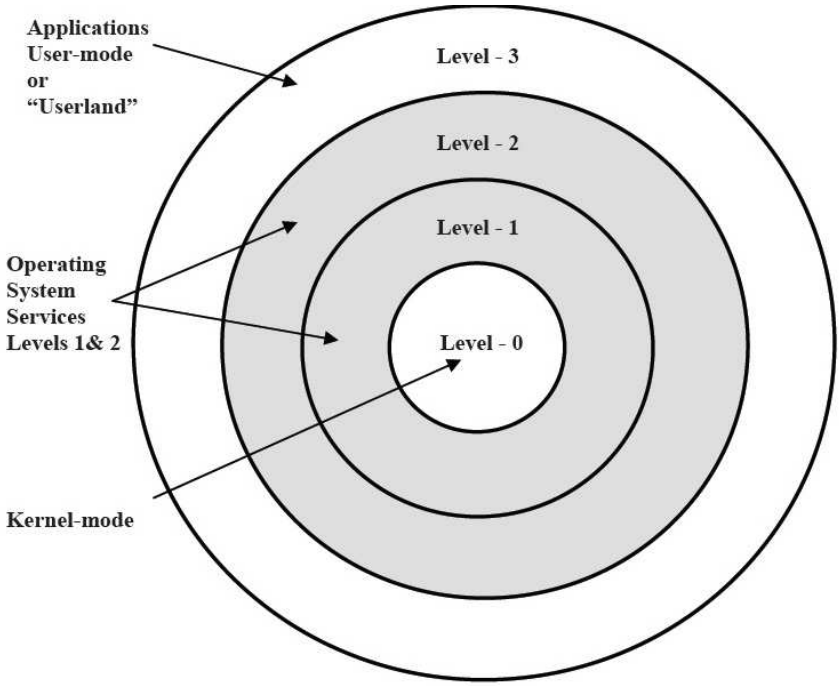


Figure 2.7: Protection Rings, Intel Developer Manual [16, 36]

we can make the amount of effort needed to compromise systems exceed that of the perceived reward then systems will be more secure because fewer attackers will attack.

One of the advantages or problems (depending on your viewpoint) with kernel rootkits is that they can cause the system to report what the attacker wishes, whether it is accurate or not [15]. This includes examples such as: incorrectly showing number and/or names of processes running, number and/or names of aircraft flying, etc.

As documented by Hoglund et al, "By using a kernel hook, your rootkit will be on equal footing with any detection software" [8]. *By placing a rootkit in the kernel we gain all of the kernel controls that would be granted to kernel level detection software.*

2.6.3 Library Rootkits. Library rootkits work like kernel rootkits although they run from the user level rather than the kernel level. By modifying libraries used by kernel calls and system calls, the attacker can redirect or modify original functionality to get the desired effects of file hiding or process hiding [15].

2.7 Summary

Classification systems for vulnerabilities/attacks and fingerprinting exist and are helpful in the defense and fixing of security vulnerabilities. However, classification only by achieved access, automation, or attack type leaves a defender wondering how to fix a security flaw. Certainly, these classification techniques give us further information on what exploit was used, but without an attack tree, it is difficult to pinpoint where and how to fix a flaw.

As discussed earlier, the time between discovery of a vulnerability and patching a vulnerability may be long enough to install a rootkit. Once a rootkit is installed it is much more difficult to remove the compromised control of the system. Once the rootkit has been installed, the patch may fix the original vulnerability which allowed access to the machine but with a rootkit now present on the machine it does not necessarily need the original vulnerability to regain access. The rootkit could be hiding any number of other attack tools that leave backdoors and other communication channels open, yet unseen.

This thesis will investigate rootkit hiding techniques (RHTs) and some publicly available rootkit detection techniques (RDTs). We then create an attack tree from which we can identify deficiencies in current detection techniques. The identified deficiencies can then be used by future researchers to increase the state of the art in computer system defense.

III. Rootkit Hiding and Finding Techniques

3.1 Chapter Overview

The main goal of a rootkit is to create an environment wherein the attacker may move about freely and stealthily. This chapter will explain some of the common stealthing techniques and give some examples of each as well as some of the current detection techniques. This thesis focuses on the stealth techniques and detection techniques although it should be noted that prevention should also be considered when protecting your computer system. Prevention techniques will become very platform and use specific. For example, in UNIX-like systems, the protection of */dev/kmem* and deactivation of kernel module support will "keep most rootkits outside of the kernel" [7].

3.2 Rootkit Stealth

In order to be stealthy, a rootkit can and should hide many things, some of which are processes and files. Three main hiding techniques are *hooking*, *patching*, and *data structure manipulation*. In general, hooking is changing the execution path of a call, patching is overwriting information in an application, and data structure manipulation is changing a data structure. Each of these hiding techniques will be discussed in further detail. Jan K. Rutkowski [56] refers to a similar separation in techniques in Figure 3.1.

Another important concept to understand when discussing stealth techniques is the difference between "hiding in plain sight" (steganography) and "hiding out of plain sight". In this thesis we use "hiding" to mean moving or covering information so that it can not be seen through the desired means (hiding out of plain sight). Steganography is another technique that can be used to obscure what the user is seeing so that they do not know that a malicious process exists because they do not perfectly recognize what they are seeing. Although steganography is normally hiding information in pictures or other files, an example of steganography in this scenario

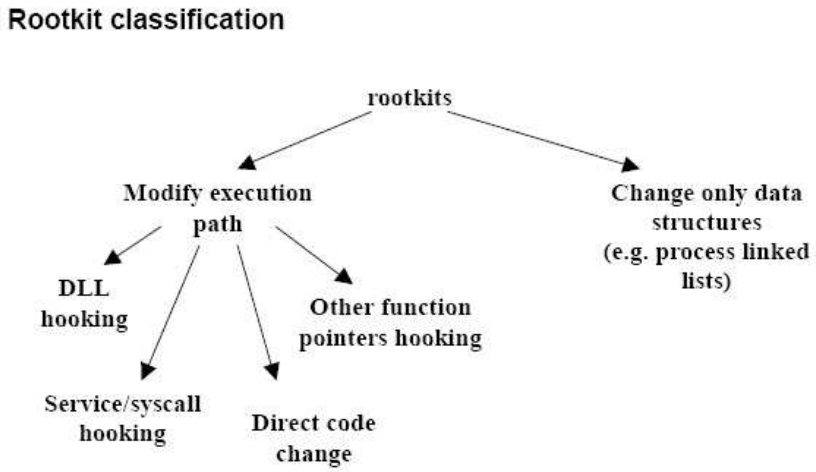


Figure 3.1: Rootkit Technology Picture [56]

would be: changing the name of a rootkit to msdirectx.sys (as is done by the FU rootkit [27]) so that it looks like a legitimate driver and thus, is hiding in plain sight.

3.2.1 Hooking. Hooking is fairly old but as quoted from Hoglund "...the rootkit-world hasn't actually moved away from system hooks. ...most systems don't run even the most basic of rootkit detection programs, so even SSDT *(system service descriptor table)* hooks are still really effective" [30].

Hooking works by changing the original execution path of some application so that the information that it receives has passed through the rootkit allowing the rootkit to scrub the data, effectively allowing the rootkit to hide itself, and anything else it chooses, from view [8]. An example, is nicely illustrated by Hoglund et al, in their book "Rootkits: Subverting the Windows Kernel" which is recreated in Figure 3.2. If we follow the diagram we see that information can be scrubbed by the rootkit, as desired by the attacker, prior to returning to the source function. First the function being rooted (source function) is overwritten with a jump to the rootkit code (the detour), next the rootkit code (the trampoline) is executed. Part of the rootkit code is to execute the overwritten data from the source function so that the proper context is in the proper registers when the original code is executed. This is

then followed by a jump to the original function plus an offset (the target) to account for the information that was overwritten. When the target function is done executing it will return the results to the rootkit (the detour/new calling function) which can then scrub the data that it desires. The rootkit will then return the scrubbed data to the original calling function.

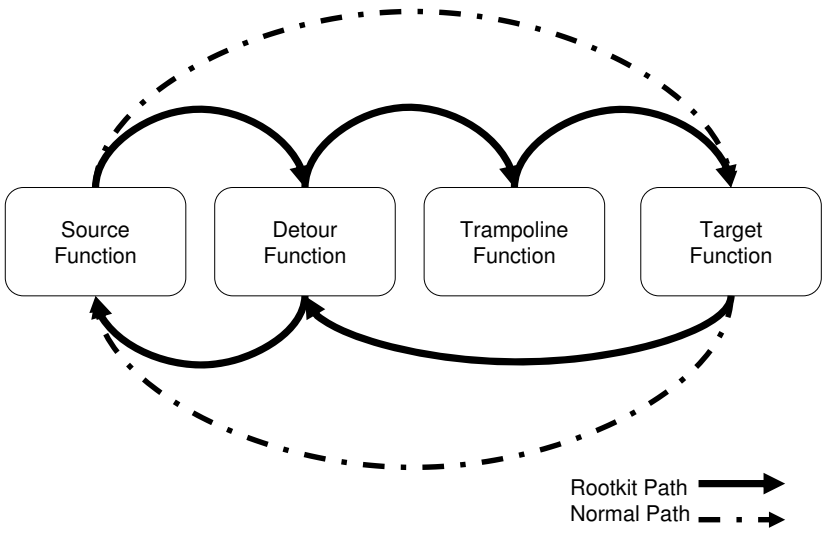


Figure 3.2: Temporal ordering of a detoured function [8]

Kernel Level Hooking - The three most common areas to hook in the Windows kernel, according to Hogland, are the System Service Descriptor/Dispatcher Table (SSDT), Interrupt Descriptor Table (IDT), and the I/O Request Packet (IRP) Function tables [8].

1. The SSDT is the table in Windows which holds a list of all system services by ID and address. As reported by Prasad Dabak et al in their book *Hooking Windows NT System Services*, Windows "system services can be considered the equivalent of system calls in UNIX" [20]. System services and system calls "represent the fundamental interface for any user-mode application or subsystem to the kernel" [20].

2. The IDT is the table which holds the interrupt identifiers and their associated addresses. An IDT exists both in Windows and UNIX-like operating systems.

In Windows, "The Interrupt Descriptor Table (IDT) is an array of 8 byte interrupt descriptors in memory devoted to specifying (at most) 256 interrupt service routines. The first 32 entries are reserved for processor exceptions, and any 16 of the remaining entries can be used for hardware interrupts. The rest are available for software interrupts" [47].

3. The I/O request packet is where information needed to process an I/O request is stored within the I/O manager and is used to represent the operation as it is processed throughout the system [53]. Because the IRP is a location that stores information about an I/O operation it becomes easy to see how this would be a valuable place to hook for a rootkit and a place to protect as a defender.

3.2.1.1 Hooking the System Service Descriptor Table (SSDT). This hooking method works on the SSDT and is not constrained to a single application. According to James Butler and Sherri Sparks in their article, *Windows Rootkits of 2005*, the SSDT is where the "actual implementation of the operating system functions are contained" [10]. SSDT hooking works by replacing the addresses of ZW* functions with the address of rootkit code ("ZW routines provide a set of system entry points that parallel some of the executive's system services. Calling a ZW routine from kernel-mode code results in a call to the corresponding system service" [17]). This gives the opportunity for the rootkit code to manipulate information that would have been returned to any calling application. ZW* functions are functions exported by the kernel for usage by other kernel functions and device drivers. When a ZW* function is called, usually by an NT* function (system call) it returns the address corresponding to the NT* function which was stored in the SSDT [8]. As can be seen in Figure 3.3, the hook simply overwrites the address of the hooked kernel function with the address of the rootkit so that any application calling the hooked function will have it's execution path pass through the rootkit giving the rootkit the opportunity to scrub or alter data.

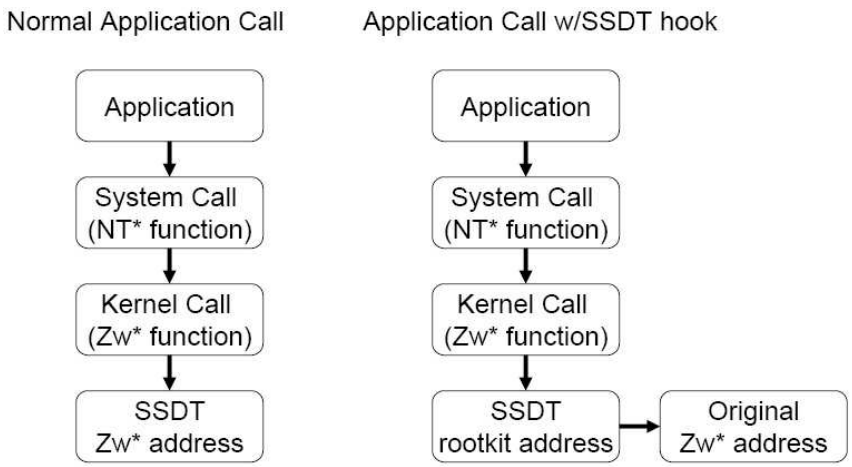


Figure 3.3: Example SSDT Hook

3.2.1.2 Hooking the Interrupt Descriptor Table (IDT). This hooking method works on interrupts and is not constrained to a single application or a single operating system. IDT hooks work fundamentally differently than SSDT hooks in that with an SSDT hook the idea is to insert the rootkit into the execution loop so that it can scrub the output of a legit function, thus returning incorrect/incomplete data to the calling application. The IDT hook, on the other hand, does not insert in the execution loop. Rather, the IDT hook breaks the loop and calls the rootkit code instead of the originally intended code as can be seen in Figure 3.4. This allows the rootkit to identify that there is a call from a particular application so that it can identify and then allow or block. This method works by replacing the address of a legitimate interrupt in the IDT with an entry to some rootkit code [8]. This causes the rootkit code to be called every time the interrupt occurs. Altering the IDT works both in Windows and in Linux simply by changing the address of a legitimate jump to the address of rootkit code in the IDT.

3.2.1.3 Hooking the I/O Request Packet (IRP) Function Tables. IRP hooks, just like IDT hooks, are not returned to so creating a hook that calls to rootkit code is necessary to alter information coming from the intended driver. An illustration of an IRP hook can be seen in Figure 3.5.

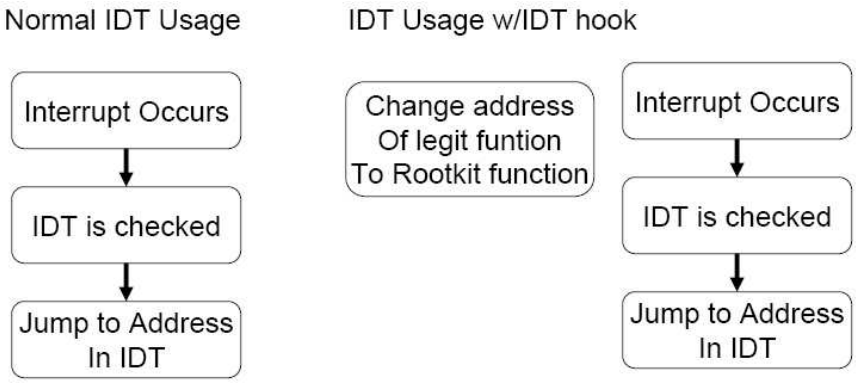


Figure 3.4: Example IDT Hook

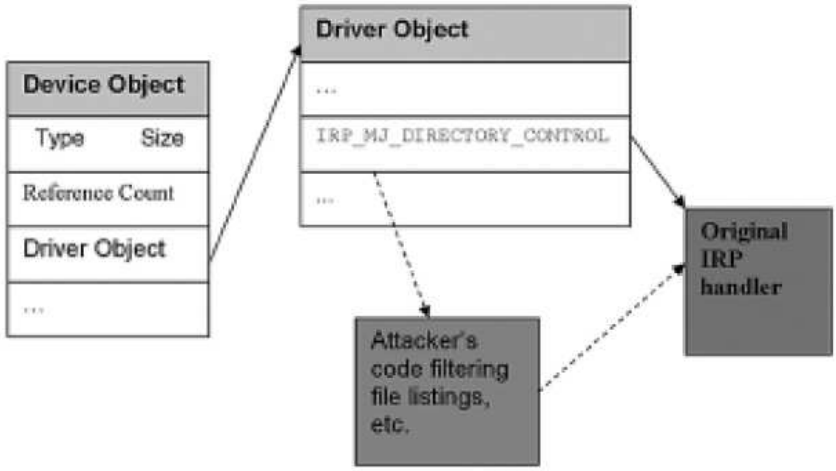


Figure 3.5: Example IRP Hook [9]

Figure 3.6, shows a normal device driver physical structure. Once hooked the diagram would change the "Device Drivers" box to "rootkit driver". It is necessary to write a completion function so that the information from the original device driver can be sanitized [8].

A common theme among the Kernel Mode hooks is that they each overwrite an address in a kernel data structure with a rootkit address and they are not application specific.

User Level Hooking can also be done from the user level or "Userland" using API hooking. API hooking is overwriting or modifying sections of files/processes used

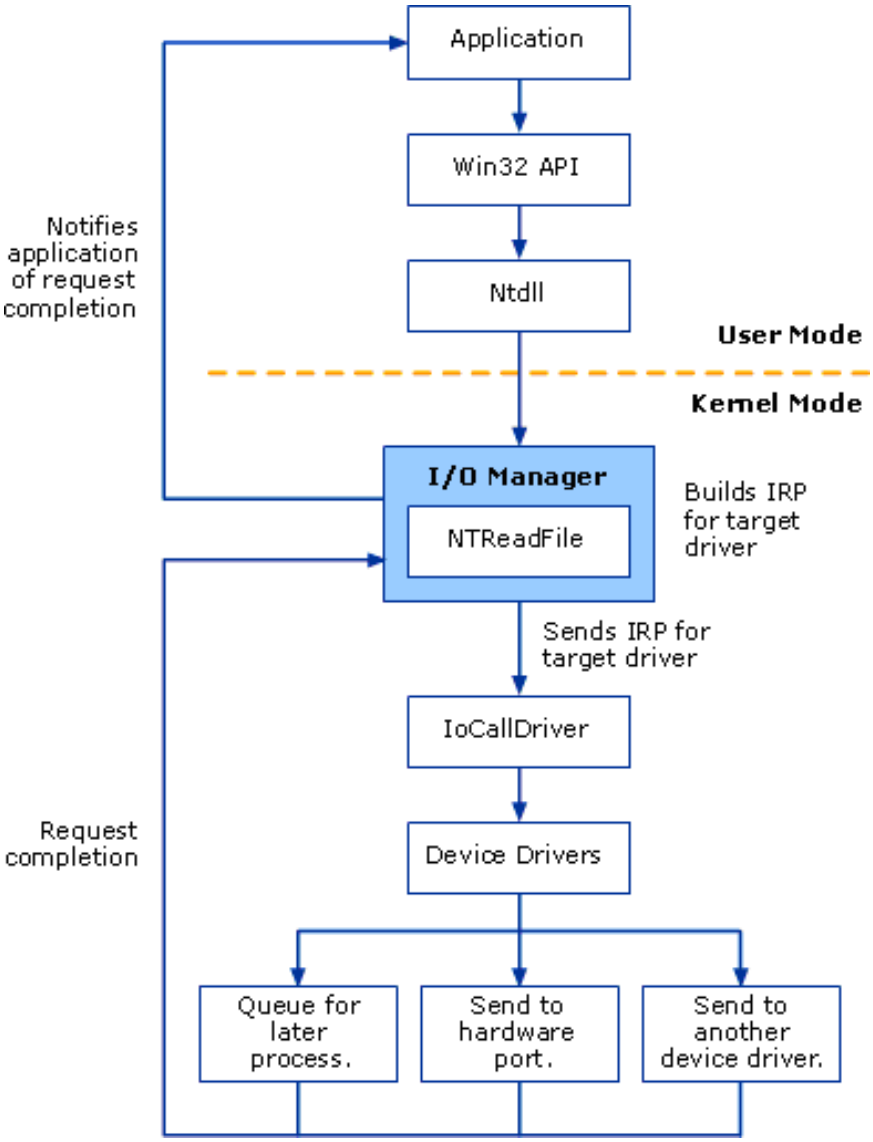


Figure 3.6: Device Driver Physical Structure, Microsoft [18]

by an application to do other than intended purposes or to report other than accurate information.

There are two common kinds of userland API hooking processes; Import Address Table (IAT) hooking, and Inline function hooking. The IAT is a data structure belongs to and resides within the address space of most application's. The IAT holds the imported addresses of functions to which the application plans to jump. Inline

function hooking is simply placing "extra" instructions into a function. The resultant execution path of inline function hooking can be seen in Figure 3.2.

3.2.1.4 Import Address Table (IAT) Hooking. IAT hooking works like SSDT hooking except it operates on the IAT table rather than the SSDT table. IAT is also a hook per application rather than a hook for all applications. The SSDT holds the addresses of system services that applications use, while each application has its own IAT. An example of IAT hooking can be seen in Figure 3.7. Normal flow would be: Application, IAT, called function. This can occur because the IAT can be overwritten to jump to the rootkit instead of the intended function. This allows the rootkit to be the caller of the intended function which gives the ability to clean/alter the returned information [8].

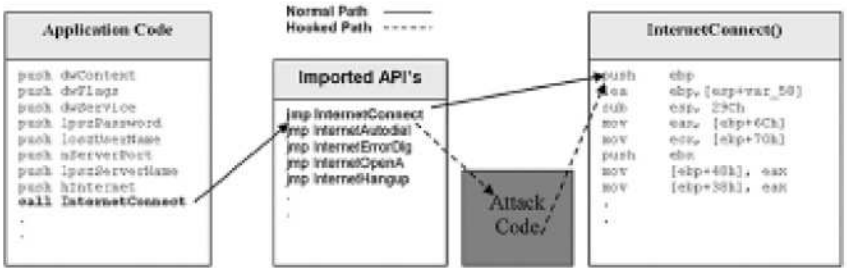


Figure 3.7: Normal execution path vs. hooked execution path for an IAT hook [9]

3.2.1.5 Inline function hooking. Inline function hooking is basically the same thing as IAT hooking except the control of the target function is taken by replacing the first 5 bytes of the target function with a jump to the rootkit function. The rootkit function then calls the target function using the previously saved first 5 bytes. In this way the target function will do its intended operation but then return the results to the rootkit rather than the calling application. The rootkit will then modify the data as needed and return the erroneous results to the calling application [8]. Figure 3.2, shows how the inline hooking would work. The actual usage of inline function hooking would look like Table 3.1, which shows the normal

preamble of a Windows function and the modified preamble. Inline function hooking is very similar to the following rootkit concept which is patching.

Table 3.1: Inline Function Hook [10]

Original Preamble	
Code Bytes	Assembly
8bff	mov edi, edi
55	push ebp
8bec	mov ebp, esp
Modified Preamble	
Code Bytes	Assembly
e9 xx xx xx xx	jmp xxxxxxxx
...	

3.2.2 Patching. Patching is overwriting a binary such that it performs in a different way than it originally performed. An example of malicious patching would be to analyze a program to find where the conditionals are located. One such conditional could be the Jump if Not Zero (JNZ) instruction which can be used to stop access to a program if the key entered is not correct/valid. This can easily be patched with the use of a hexadecimal editor. We simply search for the appropriately located hexidecimal value 75 (the JNZ assembly instruction) and change it to hexidecimal value EB (the JMP assembly instruction). This change effectively says "jump to the next instruction even if the right key is not entered" thus making the "nag screen" no longer appear. Patching is also used to improve or fix otherwise incomplete/incorrect code.

3.2.3 Data Structure Manipulation. Data Structure Manipulation (DSM) is modifying a data structure such that it no longer works in the same way. An example of DSM is Direct Kernel Object Manipulation (DKOM). Other RHTs remove data or change data while DSM is changing the structure such that the data still exists it is

just not seen or used in the same fashion. An excellent example of this technique is DKOM as implemented by FU which is described in Section 3.2.3.1,

3.2.3.1 Direct Kernel Object Manipulation (DKOM). One of the objects within the Windows kernel that can be modified is the executive process block (EPROCESS) which contains information about its associated process as well as pointers to other needed data structures [53]. In this scenario used by FU and FUTo rootkits, DKOM modifies the doubly linked list of processes such that the pointers in the list, point around the process to be hidden, effectively removing it from the list. Once the process has been removed from the list it can still continue to execute because the execution occurs through the associated threads rather than from one process to the next. This does, however, effectively remove the process from any queries that attempt to walk this linked list of processes.

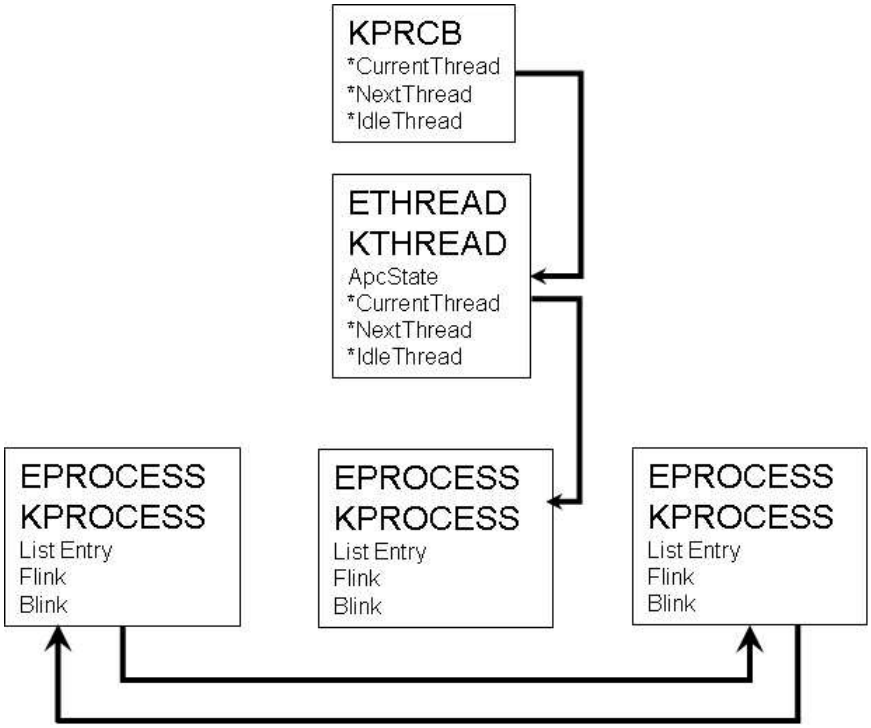


Figure 3.8: Direct Kernel Object Manipulation [8]

The removal of the EPROCESS block from the list occurs by making the forward link (FLINK) and backward link (BLINK) pointers, in the EPROCESS block of the

process to be hidden, point to each other, while making the neighboring blocks point to each other. The EPROCESS block is accessed by using the kernel processor control block to view the current thread which points to the ETHREAD block, which then points to the EPROCESS block. Each EPROCESS block is connected via a doubly linked list to other EPROCESS blocks. Figure 3.8, shows the modification of the FLINK and BLINK pointers [16].

3.2.3.2 Future DSM Targets. Data structure manipulation is one of the cutting edge rootkit techniques and as such will continue to develop. One obvious location for implementation of such techniques will be in the scheduler. Currently there is a proof of concept (used for detection) called Klister, developed by Joanna Rutkowska in 2003. Klister allows the scheduler/dispatcher lists to be read, showing the currently running threads which are used to schedule time on the processor [12,13]. Threads can then be checked to see which process they belong to, thus showing an accurate view of which processes are running regardless of hiding techniques to date such as DKOM. This RDT could be overcome at least in theory by manipulating the scheduler/dispatcher such that the desired thread(s) are hidden.

3.2.4 Virtual Machine and Virtual Memory. Virtual machine and virtual memory rootkits are some of the most cutting edge techniques. Virtual machine rootkits, such as Blue Pill by Joanna Rutkowska [54, 55], seek to turn the host OS into a virtual machine by lifting it and inserting the rootkit below as the host OS. In this way the rootkit would have access to anything and everything in the now virtual OS but the virtual OS would have no indication of the rootkit's presence. Virtual memory rootkits seek to monitor their own memory section address space such that when another process tries to read it they are redirected. An example of virtual memory rootkits is cited and explained in further detail in Section 3.2.9.

3.2.5 Hardware. Although inserting rootkits at the hardware level may be feasible, as documented by John Heasman in his paper *Implementing and Detecting a PCI Rootkit* [28], for this thesis, we will not delve into as much detail on the issue.

Summary of Stealth Techniques - As discussed earlier the three categories used by rootkits to hide are Hooking (changing the execution path), Patching (overwriting an executable), and Data Structure Manipulation (changing the data/structure of an object). These same techniques are used to attack both Windows and Linux operating systems. In Figure 3.9 we can see how the three categories fit. All software exists as data (binary code) which can potentially be overwritten or patched (inner-most circle). This data is generally in some format, object or construct of some sort (list, array, etc) which is what is hooked, in order to change the execution path (middle circle). These constructs all interact in one form or another with their environment such as the OS which is their data structure. This data structure can potentially be manipulated (outer circle). For example, changing or removing pointers to an object. It is important as a defender and an attacker to understand where all the tables with important data exist, and where control exists. Hardware certainly puts a slightly different spin on these categories but still roughly maintains these categories simply by containing these categories within a new set of hardware. For example, if a PCI card is added to a machine in order to protect the machine, the PCI card itself would have these three categories within itself. The three categories would also then exist on the same machine but outside of the PCI card thus giving two distinct locations to check.

The following sections will give some examples of the three categories of rootkits.

3.2.6 Rootkit Examples - Hooking. As previously described, hooking works by changing the original execution path of some application so that the information that it receives has passed through the rootkit allowing the rootkit to scrub the data, effectively allowing the rootkit to hide itself and anything else it chooses from view [8]. The following are some examples of hooking rootkits:

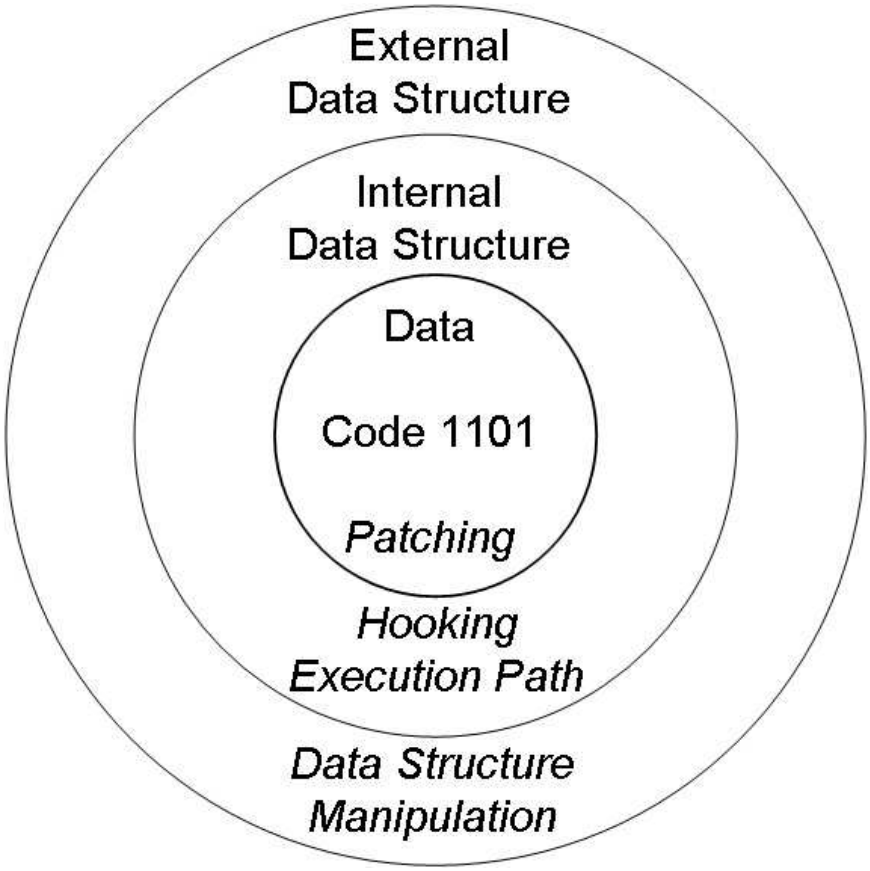


Figure 3.9: Rootkit Hiding Categories

3.2.6.1 AFX - Windows RK. Hides by hooking the SSDT as described previously. AFX, as will be shown later, can be detected by other rootkits at the same level or by checking other areas of the operating system for existence of AFX [62].

3.2.6.2 Vanquish - Windows RK. Works by hooking of API calls as well as patching. Once the API is hooked the function is patched so that the hook can subsequently be "undone" [69]. The hook no longer needs to exist once the function is patched because the function will carry out the rootkit functionality due to the patch.

3.2.6.3 Hacker Defender - Windows RK. The readme file for hacker defender describes hacker defender as follows: "... rewrite few memory segments in all running processes. ...able to hide files, processes, system services, system drivers,

registry keys and values, open ports, cheat with free disk space. ...also masks its changes in memory and hides handles of hidden processes. ...installs hidden backdoors, register as hidden system service and installs hidden system driver" [32]. This rootkit works by hooking various API functions to hook ALL processes in the system [32].

3.2.6.4 Adore BSD 0.34 - BSD RK. Adore is a kernel loaded module that is used to hide files, processes and network connections. It uses a second module to delete itself from the kernel data structures. The SSDT method described earlier is used to insert the rootkit via the overwriting of 15 system calls [7]. Subsequent versions of Adore such as Adore-NG 1.41 use the same kernel module loading but then infect other areas of the kernel such as the virtual filesystem layer such that it does not have to hide itself because it becomes part of other modules [7].

3.2.7 Rootkit Examples - Patching. Patching is overwriting a binary such that it performs in a different way than it originally performed. The following are some examples of patching rootkits:

3.2.7.1 eEye Bootrootkit - Windows RKs. Created by Derek Soeder and Ryan Permeh of eEye Digital Security, this proof of concept proves use of the boot sector to create a rootkit. It uses a hook into the interrupt 13h in order to patch the OS loader and then hooks *ndis.sys* [61] which is the Windows Network Driver Interface Specification. *NDIS.sys*, according to related forums, holds a "collection of routines that applications can invoke to perform network-related operations" [50].

3.2.7.2 SucKIT 1.3b - Linux RK. SucKIT is a Linux rootkit designed by Silvio Cesare [15] and includes mechanisms for reboots and a backdoor. This rootkit is listed in the patching section because majority of its techniques rely on patching however, as you will see many of the ideas from other techniques are used. This rootkit installs itself by doing a search on the memory of the system to find the location of *kmalloc()* (the function used to allocate memory in the kernel) and

the SSDT. Once found, the address of *kmalloc()* is placed in an unused entry of the SSDT. The entry that now exists in the SSDT is called and used to allocate kernel memory, and the rootkit is subsequently loaded into that memory space. The SSDT entry used is then overwritten to jump to the rootkit space. The /sbin/init file is also overwritten or modified in order to reload this rootkit after a reboot. However, this rootkit also copies the original file before overwriting so that when a query is made to the file, the rootkit can redirect the query to the original but renamed file. This saving and redirecting stops detection via a simple checksum. While using patching as a means for loading, this rootkit also takes advantage of the SSDT to hook 24 system calls. However, the implementation of the hook is slightly different. The SSDT is actually copied and then modified. The IDT is then hooked to have subsequent system calls jump to the modified SSDT. This stops detection of the rootkit by SSDT inspection [7].

3.2.7.3 T0rn 8 - Linux RK. T0rn hides process information by patching system libraries such as *libproc.a* in linux systems. *Libproc.a* is "used for relaying the process information from the kernelspace (via */proc* file system) to user space utilities such as */bin/ps* and *top*" [15].

3.2.8 Rootkit Examples - Data Structure Manipulation. DSM modifies a data structure such that it no longer works in the same way. The following are examples of DSM rootkits:

3.2.8.1 FU and FUTo - Windows RKs. FU and FUTo hide by using DKOM to redirect pointers within the EPROCESS block in Windows OSs as described in Section 3.2.3.1. A possible way to subvert this hiding technique is to follow each thread from the Kernel Process Control Block (KPCB) to its Ethread and subsequently to its EProcess, thus finding each process by iterating through threads in the KPCB rather than processes in the EProcess blocks. A similar concept is shown in "Klister" as previously mentioned.

3.2.9 Rootkit Examples - Virtual Memory.

3.2.9.1 Shadow Walker - Windows RKs. Shadow Walker uses virtual memory to hide itself and its malicious processes from detection. As reported by James Butler and Sherri Sparks, Shadow Walker is a proof of concept that covers three concerns in virtual memory [11]. First, it detects when something other than itself is attempting to read its memory space. This is done by identifying the difference between read, write, and execute. Differentiating between read, write, and execute (read/execute, write/execute) is accomplished by marking the page table entries (PTEs) as "non present" and hooking the page fault handler. This allows Shadow Walker to monitor access to these pages. Once it has been clearly identified which of these is being used, the RK must be able to fake the read or send back erroneous information to would-be detection utilities. If the access is a read access, Shadow Walker returns erroneous information to the application, otherwise Shadow Walker runs itself as intended. The last thing addressed by Shadow Walker is that there is little identifiable performance degradation because the increase in the number of page faults generated is minimal. [11]

Other UNIX based rootkits include: (usermode)-lkr, ark, (kernelmode)-Knark(written by Creed).

As can be seen in Table 3.2, there are many rootkits. Each rootkit studied fits into at least one of the three categories, namely hooking, patching, or DSM. However, as can also be seen in the table there are at least two rootkits which have treaded on new territory, that of VM and Hardware. Both of these new territories still fall within hooking, patching and DSM as described earlier but because of the newness were left in the table as separate entries.

3.3 Rootkit Detection

There are many techniques to detect rootkits, however, as stated elegantly by the writers of www.hxdef.org "For an attacker one security hole is enough to win

Table 3.2: Rootkit Examples

Rootkit	Hooking	Patching	DSM	VM	Hardware
AFX	X	X			
Vanquish	X	X			
Hacker Defender	X	X			
Adore/ava	X	X			
SucKIT	X	X			
Apropos	X	X			
T0rn		X			
FU			X		
FUTo			X		
deepdoor			X		
peligroso			X		
firewalk			X		
prrf			X		
phide2			X		
Shadow Walker			X	X	
Blue Pill			X	X	
eEye Bootrootkit	X	X			
HE4Hook	X				
NTRootkit	X				
NTIllusion		X			
VideoCardKit			X		X
SonyBMG XCP	X				
wootkit		X			
RK		X			

the game. One can say the role of attacker is easier. But if you want to fight the attacker you can't produce an antirootkit solution that just fixes or protects one weak point. You have to fix all those points. What's more, you can't have weak points

in that solution." [31] Rootkits are constantly evolving and improving thus rootkit detection methods must get better and more complete. There may or may not exist a single rootkit detector that can find all rootkits. James Butler and Sherri Sparks, in their article *Windows Rootkits of 2005* [12], divide rootkit detection techniques into five categories: signature based, behavioral/heuristic, crossview, integrity based and hardware detection [12].

1. Signature based detection is scanning data for a pattern which comprises a "fingerprint" that is unique to a particular entity [11].

2. Behavioral/heuristic based detection seeks to identify actions or patterns that are abnormal to system operation. These detection techniques "work by recognizing deviations in "normal" system patterns or behaviors" [11].

3. Crossview based detection uses the idea of data redundancy (the existence of equivalent data in more than one location) to find "answers" from multiple sources that should be the same in order to identify discrepancies [11]. An example of such is a child asking her mother for permission to go to the park and then asking her father permission for the same thing. The two answers should be the same; however, the difference in answers is what is exploited by the child.

4. Integrity based detection compares a known good entity with a suspected entity in order to verify the correctness/accurateness of the suspect entity. An example of such is comparing "a current snapshot of the filesystem or memory with a known, trusted baseline" [11].

5. Hardware based detection uses a piece of hardware to implement one or more detection techniques such as signature, heuristic, crossview, or integrity based detection while separating itself from the suspected operating system [11]. Hardware detection makes it more difficult for the attacking entity to corrupt the results of the hardware.

As important as each of these categories are; we have come to a different solution for categorization. If it is possible to classify, such that a taxonomy is developed, then

we will be able to move more quickly in our research to detect "all" rootkits because "all rootkits" would fit into the taxonomy. In an effort to move toward this taxonomy we propose the following categorization. We propose *detection classes*, *techniques*, and *implementations* each of which will have its subcategories.

Detection classes give categories that show ways in which it might be possible to detect. Surely there may be classes that we have not yet discovered, but to date we see these classes as: Static (Analysis/Detection based on the knowledge of characteristics possessed by an entity such as a rootkit) and Behavioral (Analysis/Detection based on the behavior or lack of absence of behavior of an entity).

Detection techniques are the subsets of detection classes which describe how the detection classes may work. Subsets of the static class are: signature (analysis/detection of an entity through identified patterns or sequences) and integrity (analysis/detection of an entity through verification and comparison of known good patterns or sequences with suspect patterns or sequences). Subsets of the behavioral class are: anomaly (analysis/detection of and entity through identification of unknown behavior) and signature (as previously defined). Each of these techniques can take on an aspect of time via a snapshot of past, present or future states.

Detection implementations are the ways in which we may implement detection techniques or combinations of detection techniques as is the case with crossview. The identified implementations to date are: crossview (the comparing of two or more inputs, whether from the same technique or various, in order to achieve detection), remote attestation (comparing "outside" information with information on a "suspect" machine), cognitive/human, hardware, software, and memory tracking (detection of anomalies via known memory behaviors).

We arrive at this categorization by analyzing detection starting with Figure 3.10. In this graph we divide detection into two areas; "What it is" (What are we trying to detect), and "What it is doing" (What is the entity, that we are trying to detect,

doing). The first branch is then followed further to identify two ways of detecting "What it is"; Is *it* there, or can we verify that *it* is not there.

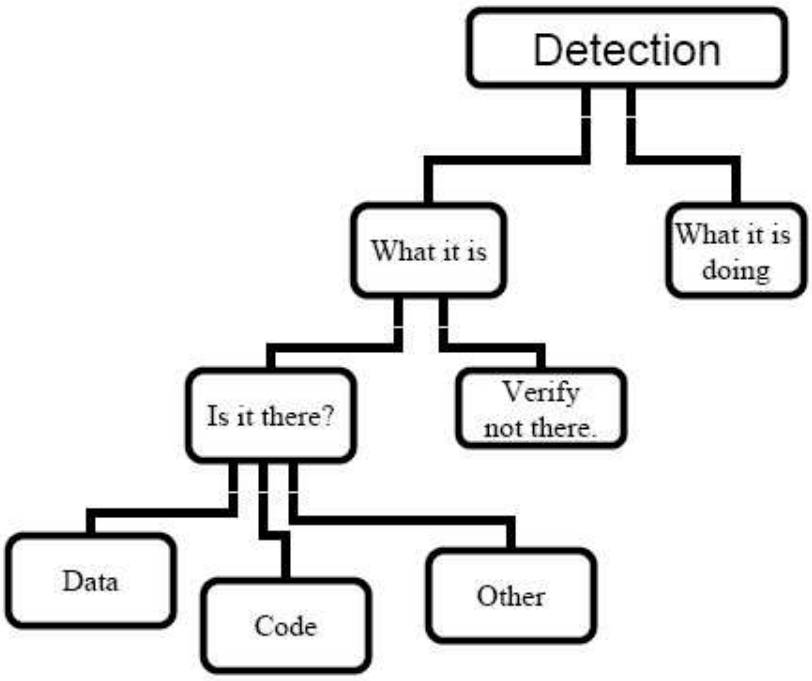


Figure 3.10: Detection Classification 1

"What it is" refers to what is the rootkit which we are trying to detect and a static view (snapshot) of either what it looked like in the past, what it currently looks like, or what it will look like in the future. This requires the knowledge of the rootkit and what it looks like at various stages. "What it is doing" refers to the actions or behavior of the rootkit or the actions and behaviors caused by the rootkit (the actions and behaviors of the surrounding system). Figure 3.11, shows the same graph with the new classes and techniques inserted. Figure 3.12 adds detection implementations and other known inputs to a system.

Other known inputs to a system that might help in detection could be intelligence (obtained from other sources, i.e., someone said there is a rootkit on my machine), witness (the data, object, or structure affected by the victim), victim (the

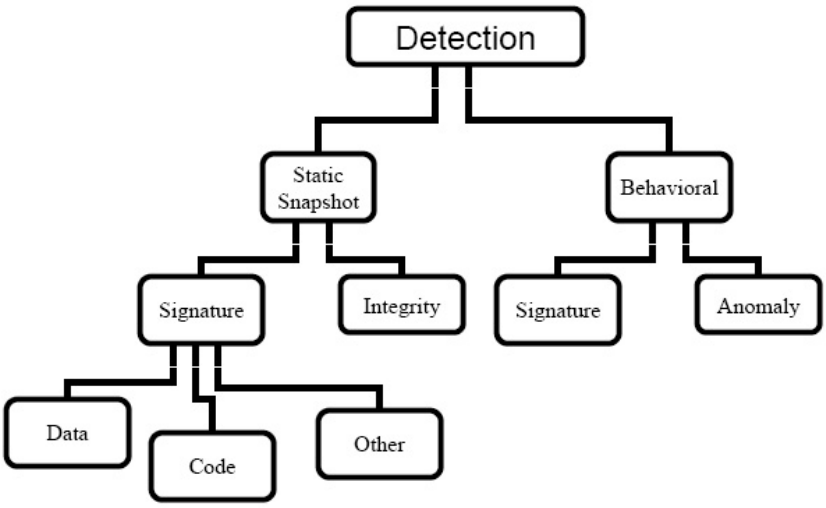


Figure 3.11: Detection Classification 2

data, object, or structure that was manipulated), attacker (the manipulator of data, object or structure), and outside sources (such as a news report or other information).

If we incorporate all of our classes, techniques, implementations and other inputs into a graph and connect them via arrows showing inputs we obtain a very busy Figure 3.12 which shows how each of the classes, techniques, and implementations interconnect.

3.3.1 Behavioral Detection Class. Behavioral based detection seeks to identify actions or patterns that are abnormal to system operation. These detection techniques "work by recognizing deviations in *normal* system patterns or behaviors" [11]. A lack of pattern or action is also a behavior, but it is a behavior of the system rather than a behavior of the entity being investigated. For example, if an application is launched that should create a particular pattern and it does not create said pattern then that is anomalous. Behavioral detection uses two main techniques: anomaly and signature. Behavioral detection can also be an input to many detection implementations such as: Crossview, hardware, software, cognitive/human, remote attestation, and memory tracking.

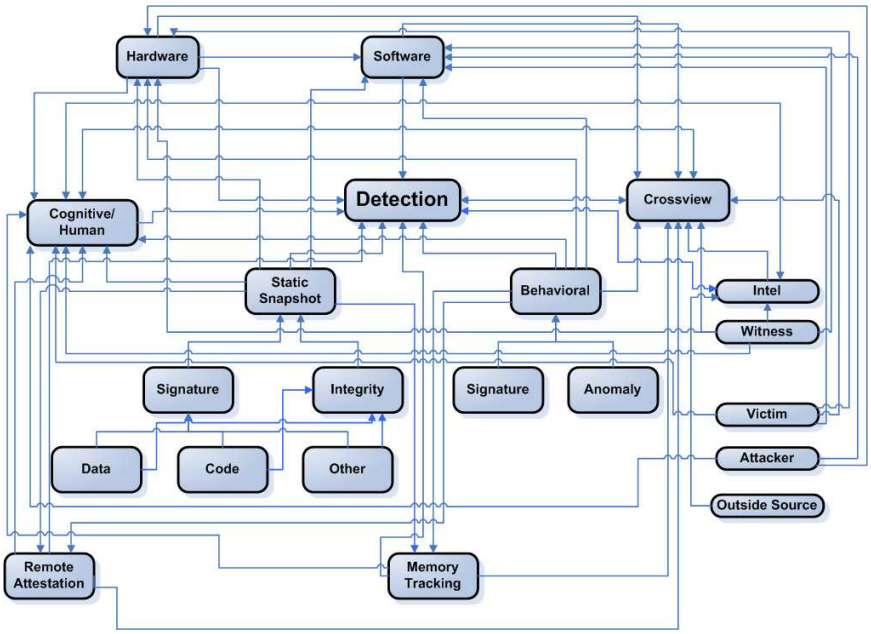


Figure 3.12: Detection Classification 3

3.3.2 Static Detection Class. The static detection class is defined by what the target entity is comprised of or its attributes, such as code. Static detection currently has two techniques, signature and integrity. Signature based detection is defined as a particular set (predefined set) of instructions or code which are to be found and acted upon (i.e., identified, blocked, removed) [23, 33]. This set may be a sequence of instructions or a pattern of instructions. Integrity based detection compares a known good entity with a suspected entity in order to verify the correctness/accurateness of the suspect entity. An example of such is comparing "a current snapshot of the filesystem or memory with a known, trusted baseline" [11]. These two techniques within the static detection class show two views, that of detection of known bad via looking for the bad and detection of bad via verifying known good.

One drawback to signature detection is the fact that it works via the blacklist (i.e., blocks a specific list of disallowed items, ie,. processes, actions, email addresses) methodology such that you must know of the attack before blocking it. The two limitations cited by James Foster of Global Security Solution Development and the glossary developed by Imperva, Data Security for the Data Center, are: 1. "They

are prone to false positives without extensive tuning", and 2. "They are not effective at detecting many unknown attacks on custom or internally developed code" [23, 33]. However, the counterpart to blacklist is whitelist (blocks all but a specific list of allowed items, ie,. processes, actions, email addresses, etc,) which is analogous to integrity based detection. By including both signature (analogy:blacklist) and integrity (analogy:whitelist) in the static detection class it seems that we have covered static analysis.

It is our hypothesis that rootkits must hide from both classes of detection, namely *Behavioral* and *Static*, in order to be *completely* stealthy. A rootkit does not necessarily need to hide from both in order to be effective but in order to obtain *complete* stealth it must address both and be at the lowest ring/highest level. The term "lowest ring/highest level" refers to the protection rings in Figure 2.7. If we do not attempt detection in all classes then a rootkit could achieve stealthiness by only implementing the class that is not challenged. For example, a particular rootkit could use metamorphic code in order to subvert signature detection and hook the integrity scan report in order to subvert the results which would in effect defeat the static detection class which shows that if behavior is not checked then we will not be able to detect such a rootkit.

One way in which some rootkits can be found, as mentioned by Hoglund et al. is by looking for an image on a system, which falls into the static detection class, "This approach is still used by most anti-virus vendors" [8]. Hoglund also suggests that "All software must 'live' in memory somewhere" [8]. which means that detection can also be implemented in the forms of guarding and scanning.

Guarding vs Scanning - Guarding is the enumeration and protection of all important "assets". This can be compared to whitelists (a specific list of allowed items, ie,. processes, actions, email addresses, etc,) and the integrity detection technique. Guarding then disallows anything not on the "approved" whitelist. Scanning, on the other hand, is looking for a particular "signature" among the unknown events and

possibly within the known events (to verify the known events' integrity). Scanning requires a "signature" which assumes apriori knowledge of the intrusion or of a particular attack method [8]. Scanning could then be compared to blacklists and the signature technique of the static class.

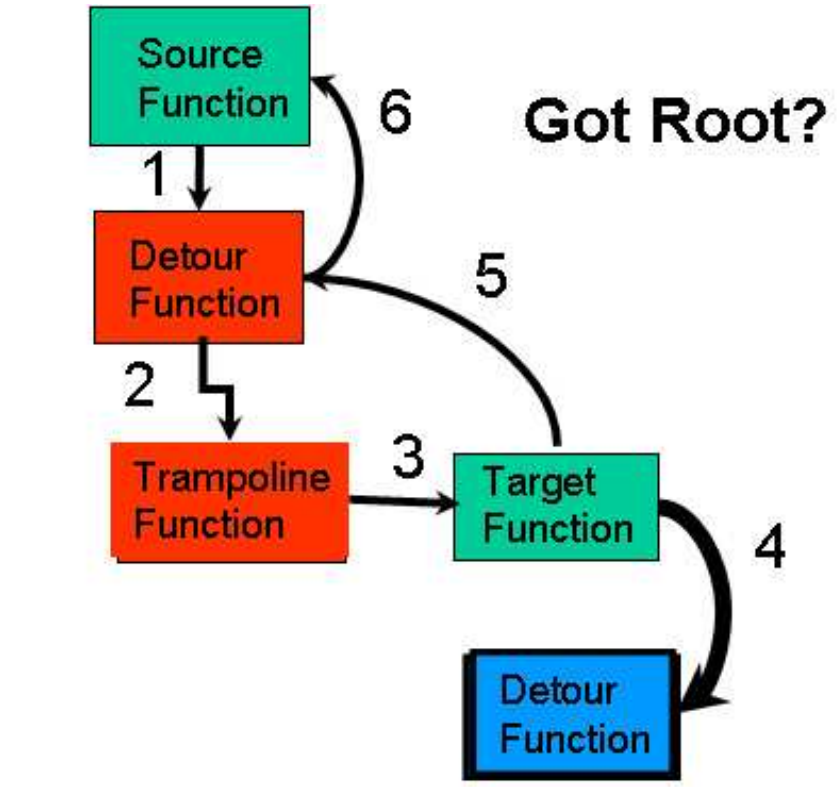
Both guarding and scanning are legitimate and useful techniques to protecting a system and when used in conjunction with one another will raise the defenses of a given system.

Rootkit techniques can be used to detect rootkits. For example, the usage of a rootkit to detect other rootkits. If we use a rootkit to report rather than to hide we may be able to subvert a malicious rootkit at the same level by creating a somewhat "secure channel" for reporting back to the user.

The lower detour function box in Figure 3.13 (step 4), shows how it might be possible to export the unmodified data of the target program to a different destination in order to maintain data integrity and compare against the modified source application data, creating a crossview. Further details of this detection technique will be discussed in Section 4.8.

3.3.3 Rootkit Detection Examples - Behavioral.

3.3.3.1 VICE. Vice detects the presence of hidden hooks by installing its own device driver to check the SSDT for pointers that do not resolve to *ntoskrnl.exe* which ("provides the Microkernel and Executive layers of the Windows NT kernel space, and is responsible for various system services such as hardware virtualisation, process and memory management") [67]. It also checks devices in *driver.ini* with the IRP table, and applications looking for IAT hooks in every DLL. This method of detection is very robust when looking for hooks of this type. However, Vice is subject to subversion through other techniques such as DSM [12].



Hoglund et al, "Subverting the Windows Kernel, Rootkits" 2006.

Figure 3.13: Trampoline Function with Modification [8]

3.3.3.2 Patchfinder. Patchfinder is a proof of concept tool created by Joanna Rutkowska. Patchfinder works by puting the x86 processor into single step mode and counting every instruction on a clean machine and comparing the instruction count to that of a potentially compromised machine. Through statistics Rutkowska claims that rootkit detection can occur because the application of a histogram shows shifts in peak instruction counts on an infected system versus a non infected system. Differently stated, a clean system will have a peak in instruction counts at a particular spot in execution and an infected system will have a different or shifted peak. However, the height of the peak may vary, normally, because of the different paths of execution that could exist [12].

3.3.3.3 System Virginity Verifier. System Virginity Verifier, created as another proof of concept by Joanna Rutkowska, works much like VICE by "comparing important system libraries and drivers on disk with their corresponding loaded images in memory" [12].

3.3.3.4 Copilot. Copilot is a hardware detection tool in the form of a PCI card which implements behavioral and signature based detection techniques on key tables and functions. Copilot effectively solves (at least for now) the need to have a secure channel for reporting back to a user by maintaining all of the processing of information on its own CPU using Direct Memory Access (DMA) to scan for rootkits, and its own network interface to securely send the information to the requestor [12].

3.3.4 Rootkit Detection Examples - Signature.

3.3.4.1 Klister. Klister finds processes by iterating through the scheduler to find all threads that are scheduled and comparing that list to a higher level view of running processes thus allowing a crossview to find discrepancies. A crossview uses two or more views of the same data and compares the results in order to identify discrepancies. Klister is a proof of concept for the Windows 2000 platform created by Joanna Rutkowska [12]. It is claimed to be subvertible at least in theory by changing the scheduler code or by using virtual machine technology. However, both of these subversion techniques not only raise the level of difficulty for rootkit creation but have an inherent problem in that they increase processor workload, thus allowing detection via delay issues.

3.3.4.2 Rootkit Revealer. Rootkit revealer seeks to find persistent rootkits (those that remain between reboots) by comparing the registry hive ("comprises a set of files, called hives, that are stored on the hard drive" [41]) and the filesystem to identify files that do not belong [12]. This method of detection can easily be subverted through redirection of queries to the file system files and queries

to the registry. According to the technichal paper, *An Analysis of Forensic Tools in Detecting Rootkits and Hidden Processes* by Todd et al [3], Rootkit Revealer was able to detect AFXRootkit and Hacker Defender.

3.3.4.3 Strider GhostBuster. Strider Ghostbuster seeks to find rootkits by comparing high level API calls and a manually parsed version of the filesystem to identify files that do not belong [12]. As with many of these detection techniques, this method of detection can easily be subverted through use of hooking and redirection. Therefore, it is important that the information obtained is verified to be correct possibly through the use of trusted computing methodology.

3.3.4.4 Tripwire. Tripwire accomplishes detection through integrity checking of the disk. In order for this to succeed the user must take a clean view of their disk prior to operation and then any subsequent checks are compared against the clean view. Each view or check uses a Cyclic Redundancy Check (CRC) hash value to record the state of the disk. Any variations to the CRC in future checks from the original check constitutes a flag or detection. Subversion of this technique is accomplished through in memory rootkits as opposed to on disk (persistent) rootkits [12].

3.3.4.5 Blacklight. Blacklight works by querying every possible Process Identification (PID) number, which they call Process Identification Brute Force (PIDB), and comparing the results with a crossview of a higher level call to mark any discrepancies as hidden. The following quote from an article published on uninformed.org by Peter Silberman and C.H.A.O.S. clearly describes how Blacklight works:

> "Now we have a complete picture of how Blacklight detects hidden processes: Blacklight starts looping through the range of valid process IDs, 0 through 0x41DC. Blacklight calls OpenProcess on every possible PID. OpenProcess calls NtOpenProcess. NtOpenProcess calls PsLookupProcessByProcessId to verify the process exists. PsLookupProcessByProcessId uses the PspCidTable to verify the processes exists. NtOpenProcess calls ObOpenObjectByPointer to get the handle to the process. If Open-

> Process was successful, Blacklight stores the information about the process and continues to loop. Once the process list has been created by exhausting all possible PIDs. Blacklight compares the PIDB list with the list it creates by calling CreateToolhelp32Snapshot. CreateToolhelp32Snapshot is a Win32 API that takes a snapshot of all running processes on the system. A discrepancy between the two lists implies that there is a hidden process. This case is reported by Blacklight" [63].

According to the technichal paper, *An Analysis of Forensic Tools in Detecting Rootkits and Hidden Processes* by Todd et al [3], BlackLight was able to detect AFXRootkit, Hacker Defender, Vanquish, FU and FUTo.

3.3.4.6 Ice Sword. Ice Sword, was created by a Chinese programmer by the alias of pjf_ [40]. It is believed to function using the same techniques as Blacklight for process detection [63]. However, an apparent advantage Ice Sword has over it's counterpart is that it is more robust and includes capabilities to detect "hidden processes, services, drivers, files, ports, and registry settings" [3]. According to the technichal paper, *An Analysis of Forensic Tools in Detecting Rootkits and Hidden Processes* by Todd et al [3], Ice Sword was able to detect AFXRootkit, Hacker Defender, Vanquish, FU and FUTo.

3.3.4.7 GMER. GMER scans for hidden processes, threads, modules, services, files, alternate data streams, registry keys, SSDT hooks, IDT hooks, IRP calls, and inline hooks. GMER also monitors creation of processes, driver loading, library loading, file functions, registry entries, and TCP/IP connections [48].

The area of rootkit detection has grown considerably and there are a very large number of detectors. Due to the high number of detectors available, not all were able to be covered due to time contraints. However, a list of some that were found are listed in Table 3.3. Some were found mentioned in various articles but not described fully and some were found at sites such as majorgeeks.com [3].

Table 3.3: Rootkit Detection Examples

Rootkit Detector	Static	Behavioral
Klister (Rutkowska)	X	
Rootkit Revealer (Sysinternals)	X	
Strider Ghostbuster (Microsoft)	X	
Tripwire	X	
F-Secure Blacklight (F-Secure)	X	
Ice Sword (XFocus)	X	
Patchfinder		X
Vice		X
System Virginity Verifier (Rutkowska)		X
CoPilot		X

Other Rootkit Detectors include: RKDetector, find hidden service, Flister, Kill hide services, Kernel hidden process/module checker, modGreper, RegDatXP, TaskInfo, and bluestone.

Windows RK Detectors include: Aries Sony Rootkit Remover(lavasoft), Archon Scanner(x-solve), AVG AntiRootkit(Grisoft), Avira Rootkit Detection(Avira), Dark Spy(CardMagic and Wowocock), Helios(MIEL e-Security), HiddenFinder(Wenpoint), HookExplorer (iDefense), Panda Anti-Rootkit Tucan (Panda Software), Process Master (Backfaces), Rootkit Detective (McAfee Avert Labs), Rootkit Buster (Trend Micro), RootKit Hook Analyzer (Resplendence), RootkitShark (Advances.com), Rootkit Uncover (Bit Defender), Rootkit Unhooker (UG North), SEEM (Al, nunki), Sophos Antirootkit (Sophos), and Unhackme (Greatis).

Linux/BSD RK Detectors include: chkrootkit(Murilo and Jessen), Zeppoo (Zeppoo), and Rootkit Hunter (Boelen).

Mac RK Detectors include: OS X Rootkit Hunter (Christian Hornung).

Summary - The detection classes, techniques, and implementations with their associated examples all give us a better view into how rootkits can be detected and

where we may find potential deficiencies. The following chapter will show attack trees and defense trees as examples of how rootkits hide and are detected in order to help further identify what these deficiencies may be. It also illustrates via ideas and experimentation some ways to further the state of the art.

IV. Experimentation and Results

4.1 Chapter Overview

In this chapter, in order to further explain and classify rootkit hiding techniques, we develop an attack tree and a defense tree which will help to identify deficiencies in current detection, help focus future research of hiding and detection, as well as important concepts for defense. We also explain and define an experiment which shows how a rootkit can be used to defend against other rootkits of the same type in order to springboard future defensive techniques.

4.2 Rootkit Attack Tree

In Figure 4.1 we have created an example of how an attacker might achieve a degree of stealth via following one of the branches of this attack tree through its associated OR gates. We note that in order to obtain a degree of stealth the attack need only pick one of the shown attack paths. For example, the FU rootkit [26] modifies the SSDT which provides some stealth because the SSDT is relied upon by many programs to provide information about existing processes.

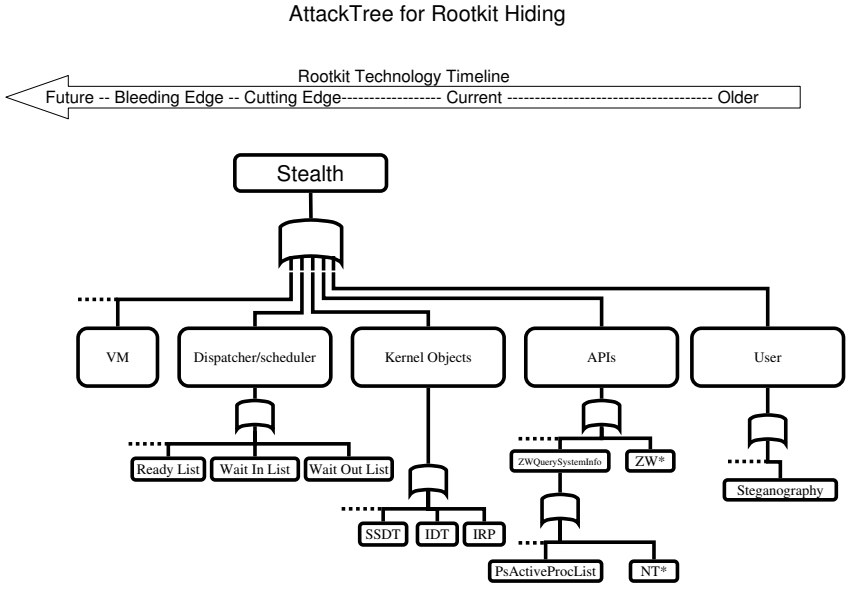


Figure 4.1: Attack Tree 2: Rootkit Hiding Techniques

From Figure 4.1 a hierarchical view of complete stealth can be derived and is shown in Figure 4.2. Thus, for complete stealth an attacker must descend to the lowest level of stealth. However, the lowest level of stealth needed is dictated by the level of defense being implemented. Furthermore, descending to the lowest level without regard to the higher levels is also not sufficient; the attacker must keep in mind the 30,000 foot view in order to maintain stealthiness. For example, if an attacker descends to the lowest level of stealth (perhaps using a VM technique such as Blue Pill) but forgets to hide from both the signature and behavioral detection classes with all of their techniques and implementations then they can still be theoretically detected. In this case, although popular belief and albeit, very good analysis, would suggest that Blue Pill would be "...virtually '100% undetectable'!" [54], this does not account for static or behavioral analysis from a hardware implementation and possibly others.

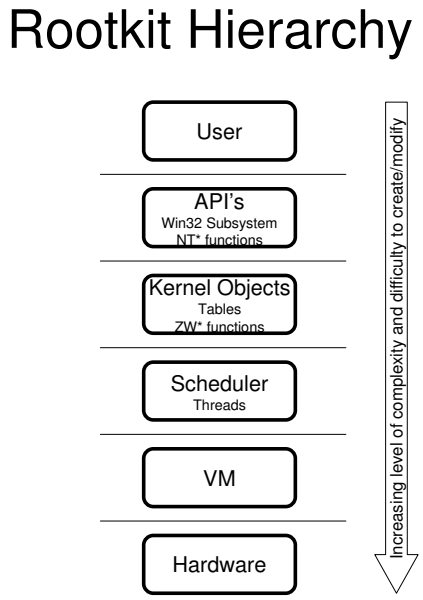


Figure 4.2: Rootkit Hierarchy

4.3 Rootkit Defense Tree

A defense tree is the counterpart to an attack tree, and is used simply to identify what needs to be defended in a computer system. In order to have complete protection, all of the branches of the tree must be checked as can be seen in Figure 4.3 which uses AND gates to depict that all branches must be addressed.

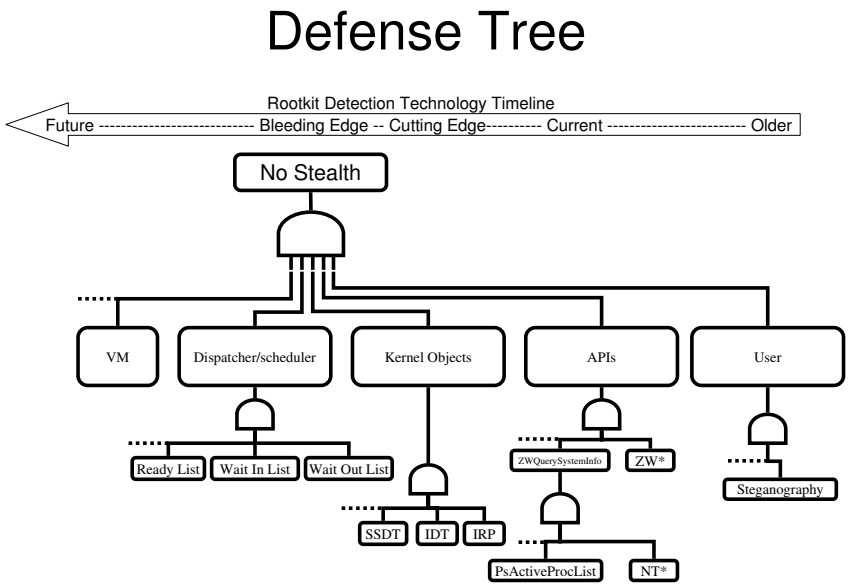


Figure 4.3: Rootkit Defense Tree

However, some branches can potentially be checked via checking other branches or combinations of branches. An example of defense tree usage is given in Figure 4.4, with Klister. By using a view of the API branch and a view of the scheduler/dispatcher (the part of the operating system that decides what is to run next and for how long based on priorities), Klister is able to effectively discover what is occuring in one branch without directly searching it because the dispatcher should hold an accurate list of what is running which can be compared against what "should be" accurate in the API list. Any discrepancies can then be shown as hidden processes. This technique cuts out the DKOM technique used by FU and FUTo by working around it. However, as mentioned earlier, all branches should still be checked because if the rootkit gets below the dispatcher then this technique, as clever as it is, also fails.

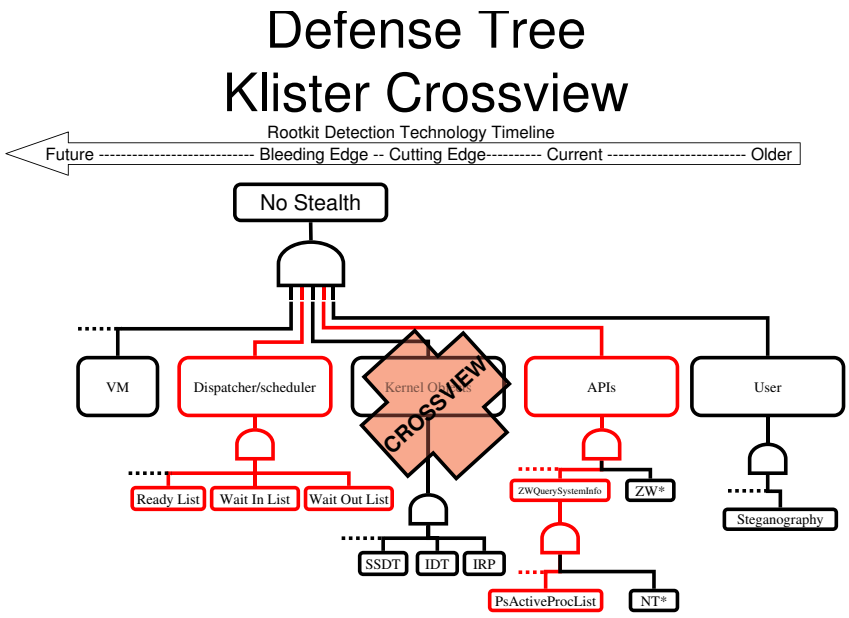


Figure 4.4: Rootkit Defense Tree:Klister Example

4.4 Furthering the state of the art

In order to increase the state of the art in finding techniques we need to continue to innovate and create new ways to detect and "win" the arms race. One possible way to increase detection abilities is to use the available hiding abilities. To this end, we have created the following experiment which uses a hooking rootkit to find other hooking rootkits. In Section 4.5 we outline the System Under Test (SUT), which is then followed by an experiment overview in Section 4.6, setup in Section 4.7, and outcomes in Section 4.8.

4.5 System Under Test

The system boundaries for this particular research must be limited in order to be manageable and contribute information to the body of knowledge on rootkits. For this research we make the following boundaries:

4.5.1 Base System. Dell Latitude D510, Intel Celeron(R)M 1.40GHz processor, 1G Ram, Windows XP Professional version 2002; service pack 2; fully patched as of 8/21/2006.

4.5.2 Tested System. VMware Workstation version 5.5.1 build19175 with Windows XP Professional version 2002; service pack 2; unpatched.

4.5.3 Software. Rootkits: HideProcessHookMDL, Modified HideProcessHookMDL, Hxdef100r, AFX2005, and Fu. Support Software: Procexp.exe, dbgview.exe, calc.exe

4.6 Experiment Overview: Rooted Rootkits

It is our hypothesis that if a rootkit hides using a particular method that it can also be modified to find other rootkits, of the same or "lesser" classes, using that same method. The first experiment simply checks to see if a rootkit "HideProcessHookMDL" can be used to detect other rootkits that use the same and similar methods of hiding.

HideProcessHookMDL (HPH) hides by hooking the SSDT and scrubbing the content of the returned linked list for anything that has _root_ in its name [25].

To detect the ability to find other rootkits we modified HideProcessHookMDL such that it still hooks the SSDT but does not hide any processes, rather, it sends a debug print statement which can be seen using Dbgview.exe [51] from Sysinternals. Due to the technique used to hide via a hook into the SSDT, nothing is readily seen as output until a call is generated looking for processes. Therefore, in order to more closely monitor and show results for this experiment we used another tool called procexp.exe [52] also from Sysinternals, because it continually calls for a list of running processes. This also gives us the ability to see which processes are running from the application perspective (procexp.exe) so that we can compare those with what the modified hook (via the debug statements in dbgview.exe) sees running. It

is also necessary to name a program with _root_ in the name. For these experiments we will simply make a copy of calc.exe and name it _root_calc.exe. The final piece of software needed to make this experiment complete is InstDriver.exe [29] which is used to install the modified HPH.

4.7 Experiment setup

Install Modified HPH on a clean system (Windows XP used). Start HPH. Start Debugview. Start Process explorer. Start calc.exe and _root_calc.exe. Install and start other hooks. Compare output of application and modified hook.

4.8 Outcomes

4.8.1 Experiment 1 Modified HPH vs unmodified HPH. This experiment verified that HPH was working because the process (_root_calc.exe) did not show up in the process explorer. However, it showed that this particular hook could detect itself because the hook did show up in debugview.

4.8.2 Experiment 2 Modified HPH vs Hxdef100r. This experiment verified that both HPH and hxdef100r were working because the process (hxdefcalc.exe) did not show up in the process explorer. However, it did show up in debugview. This illustrates that the modified HPH was able to defeat hxdef100r.

4.8.3 Experiment 3 Modified HPH vs AFX2005. This experiment verified that both HPH and AFX2005 were working because the process (_root_calc.exe) did not show up in the process explorer. However, it did show up in debugview. Which furthermore illustrates that the modified HPH was able to defeat AFX2005.

4.8.4 Experiment 4 Modified HPH vs Fu. As expected this experiment showed that Fu was able to continue hiding processes even with the modified HPH installed because Fu uses a different hiding technique discussed previously called

DKOM. This experiment further illustrates the "arms race" that exists with rootkits versus rootkit detectors.

4.9 Metrics

Metrics in this experiment only had two outcomes, success or failure. The results are summarized in Figure 4.5.

TEST RESULTS

Modified HPH vs ...	Success	Failure
Unmodified HPH	X	
Hxdef100r	X	
AFX2005	X	
FU		X

Figure 4.5: Test Results

In this chapter we have shown how attack trees and defense trees can be developed in order to identify holes in detection and also ways to detect in all branches even when a particular branch has been otherwise compromised.

V. Summary

5.1 Conclusion

This thesis has shown the importance of understanding rootkit stealth techniques and rootkit detection techniques. Many of the current technologies for each stealth technique and detection class were illustrated in Chapter 3. Rootkit stealth techniques can be examined using a graphical representation called an attack tree. The attack tree and its counterpart the defense tree developed in this thesis show what categories of techniques touch the base of the tree and shows to successfully defend against all rootkits, each and every category of rootkits must be at least addressed. It is also important to remember that this is rootkit research is very comparable to an arms race in that, what works today may not work tomorrow because the attackers will invent new ways of getting in that we as defenders have not yet thought of and visa versa.

It was experimentally shown that a rootkit can successfully find other rootkits. This research explored ways in which rootkits hide and also how they can be detected. These hiding and finding techniques were used to create an attack tree from which we can identify deficiencies in current detection techniques. Detection categorizations were also refined.

The following section outlines some ideas for future research.

5.2 Future Research

5.2.1 Rootkit Detection concept: Screen Sweeping. *Screen Sweeping* is removing data and/or results from the computer system user's view. If a rootkit is installed on a machine and a rootkit detector successfully identifies that there is a rootkit present, the next logical step is to alert the user. If a rootkit can stop or modify the detection report before it is sent to the computer screen then the user will not know that a rootkit is present even if the detection tool was initially successful.

Screen Sweeping, suggests that in order to detect perfectly we must not only detect but create a "secure channel" from the level of detection to the screen. This

"secure channel" or "Secure I/O" [38] is one of the concepts that is currently being addressed by work in Trusted Computing, which is explained in Section 2.4.

5.2.2 Rootkit Detection concept: Memory Tracking. Every system has a finite amount of memory. We understand how memory is used and the algorithms/applications that choose which memory will be used. If we can mark or track which memory is being used by our known system and which memory is simply "trash" then by carefully tracking the memory we should be able to see writing anomalies and thus detect other potentially malicious software on our systems. For example, if we know that we have 100 memory locations in our system and locations 1-10 are being used, then we should know that upon installation of a new program or other such action that memory location 11 should be written used (location of the next memory location would be algorithm/software dependent but should be understood). If upon installation we tracked memory and noted that the chosen location for installation was not 11 but rather some other location then we can set a flag as anomalous and further investigate. What is at memory location 11? Is it a bad sector? Is it malicious code?

5.2.3 Research Questions.

1. Do multiple detection methods (ie, crossview) need to be used or can a trusted channel be created that is sufficient to detect and report all rootkits?
2. Is the first "complete" rootkit installed on a machine truly the winner? Is there no other detection/removal option if a rootkit addresses each category and each level?
3. Can I detect hardware scans and feed them faulty information or hide from them?
4. Are there any other classes of detection other than static and behavioral?
5. Are there any other implementations of the static detection class other than signature and integrity?

6. Are there any other implementations of the behavioral detection class other than anomaly and signature?
7. Are there any other detection techniques other than crossview, hardware, cognitive/human, software, remote attestation and memory tracking?
8. Explore ways to implement memory tracking.

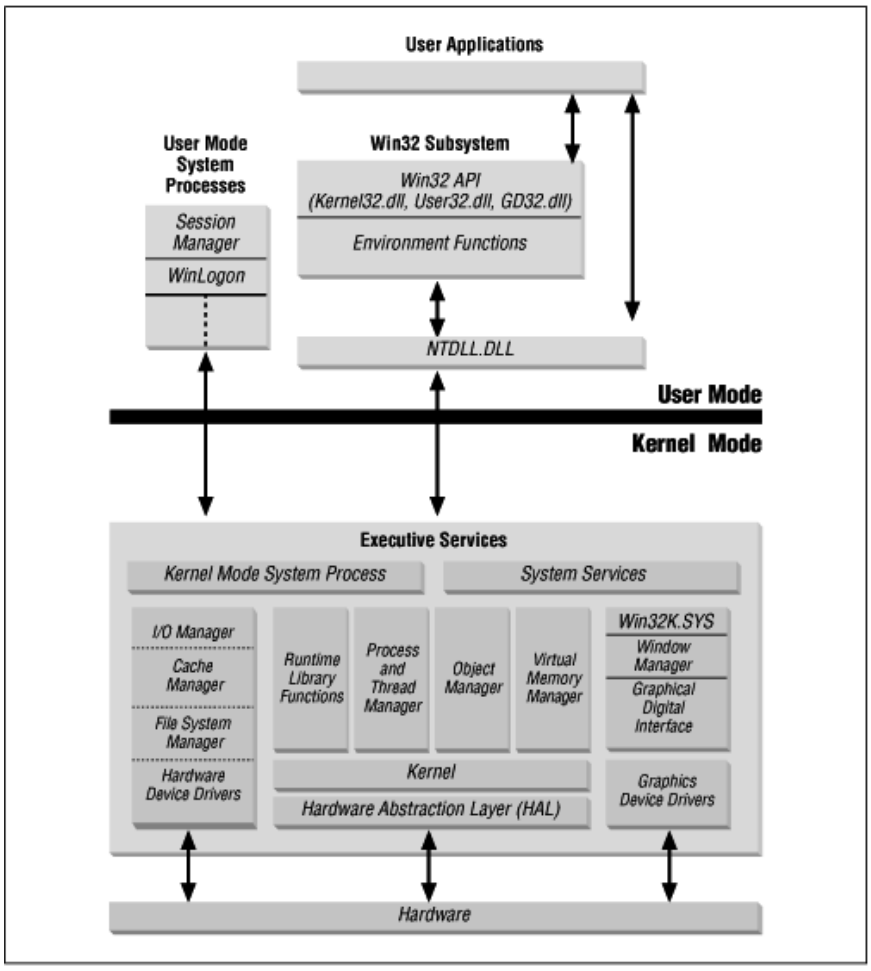


Figure A.1: Windows Architecture [19]

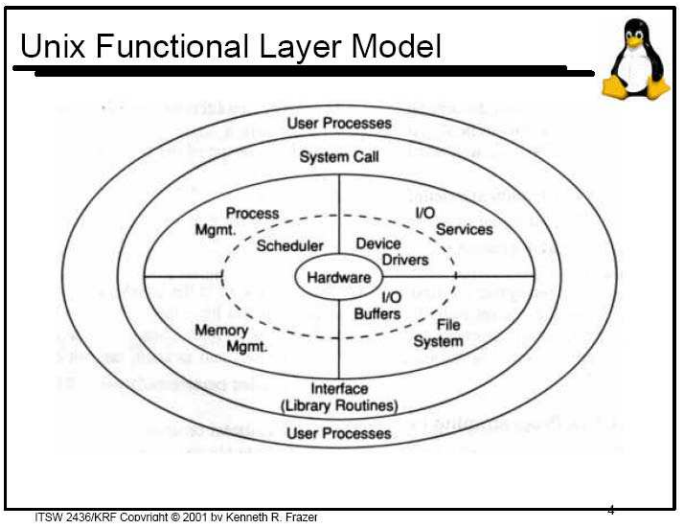


Figure B.1: UNIX Architecture [24]

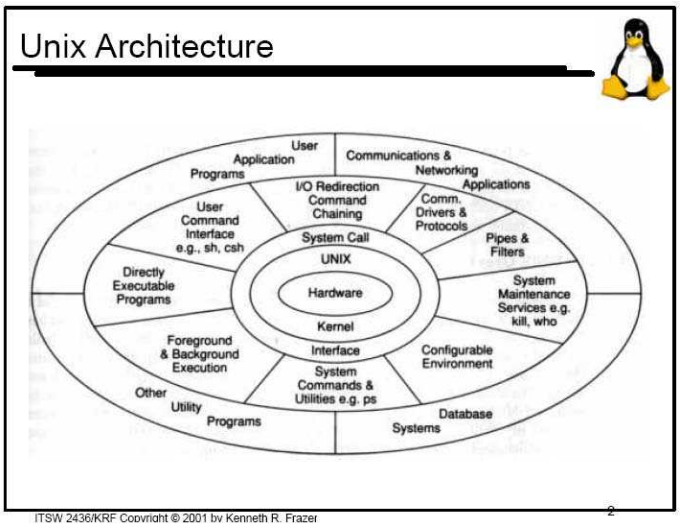


Figure B.2: UNIX Architecture [24]

Bibliography

1. Adamantix.org. "Adamantix", 2007. URL http://www.adamantix.org/.

2. of the Air Force, Department. "Air Force Instruction 33-138", Nov 2005. URL http://www.e-publishing.af.mil/pubfiles/af/33/afi33-138/afi33-138.pdf.

3. Benson, Joshua A, Timothy P Franz, Michael R Stevens, Adam D Todd, Gilbert L Peterson, and Richard A Raines. *An Analysis of Forensic Tools in Detecting Rootkits and Hidden Processes*. Technical report, Sep 2006.

4. BERINATO, SCOTT. "PATCH AND PRAY", Aug 2003. URL http://www.csoonline.com/read/080103/patch.html.

5. Bishop, Matt. "Computer Security Art and Science", 2003.

6. Bruning, Max. "A Comparison of Solaris, Linux, and FreeBSD Kernels", Oct 2005. URL http://www.opensolaris.org/os/article/2005-10-14_a_comparison_of_solaris__linux__and_freebsd_kernels/.

7. Bunten, Andreas. *UNIX and Linux based Rootkits Techniques and Countermeasures*. Technical report, 2004. URL http://www.first.org/conference/2004/papers/c17.pdf.

8. Butler, James and Greg Hoglund. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2006.

9. Butler, James and Sherri Sparks. *Spyware and Rootkits: The Future Convergence*. Technical report, Dec 2004. URL http://www.usenix.org/publications/login/2004-12/pdfs/spyware.pdf.

10. Butler, James and Sherri Sparks. "Windows rootkits of 2005, part one", 4 Nov 2005. URL http://www.securityfocus.com/infocus/1850.

11. Butler, James and Sherri Sparks. "Windows rootkits of 2005, part two", 17 Nov 2005. URL http://www.securityfocus.com/infocus/1851.

12. Butler, James and Sherri Sparks. "Windows rootkits of 2005, part three", 5 Jan 2006. URL http://www.securityfocus.com/infocus/1854.

13. Butler, Jamie and Greg Hoglund. "VICE - Catch the hookers!", 2005. URL http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf.

14. Center, CERT Coordination. "CERT CC Statistics 1988-2006", 7 June 2006.

15. Chuvakin, Anton. *An Overview of UNIX Rootkits*. Technical report, iDEFENSE Labs, Feb 2003. URL http://mitglied.lycos.de/dlwhitepapers/rootkits.pdf.

16. Claycomb, Craig A. *Analysis of Windows Rootkit Detection Tools (FOUO)*. Ph.D. thesis, AFIT, 2006.

17. Corp, Microsoft. "Kernel-Mode Driver Architecture: Windows DDK", 2002.

18. Corporation, Microsoft. "Device Driver Physical Structure", 2006. URL http://technet2.microsoft.com/WindowsServer/en/library/2e81a334-ece5-4210-815a-6a2ea33f61151033.mspx?mfr=true.

19. Corporation, Microsoft. "Windows Architecture", 2006. URL http://www.microsoft.com/technet/archive/ntwrkstn/evaluate/featfunc/winarch.mspx?mfr=true.

20. Dabak, Prasad, Milind Borate, and Sandeep Phadke. *Hooking Windows NT System Services*. M&T Books, Oct 1999. URL http://www.windowsitlibrary.com/Content/356/06/2.html.

21. developer.apple.com. "Developer Connection", Nov 2006. URL http://developer.apple.com/documentation/Darwin/Conceptual/KernelProgramming/security/chapter_3_section_7.html.

22. ESISAC. "American Electric Power Attack Tree Methodology", 6 June 2006. URL http://www.esisac.com/publicdocs/assessment_methods/AppG_AEP_ATM.pdf.

23. Foster, James C. "IDS: Signature versus anomaly detection", 2005. URL http://searchsecurity.techtarget.com/tip/0,289483,sid14_gci1092691,00.html.

24. Frazer, Kenneth R. "UNIX Architecture", 2001. URL http://home.earthlink.net/ krfrazer2/Unix_Architecture.pdf.

25. fuzen_op. "HideProcessHookMDL", 2005. URL http://www.rootkit.com/vault/fuzen_op/HideProcessHookMDL.zip.

26. fuzen_op. "FU", 2006. URL http://www.rootkit.com.

27. fuzen_op. "FU_Readme.txt", 2007. URL https://www.rootkit.com/vault/fuzen_op/FU_README.txt.

28. Heasman, John. *Implementing and Detecting a PCI Rootkit*. Technical report, NGSSoftware Insight Security Research (NISR), 15 Nov 2006. URL http://www.ngssoftware.com/research/papers/Implementing_And_Detecting_A_PCI_Rootkit.pdf.

29. Hoglund. "InstDriver", 2003. URL https://www.rootkit.com/vault/hoglund/instdvr.zip.

30. Hoglund. "OpenRCE Article Comments: FUTo", 2006. URL http://www.openrce.org/articles/view_comments/19.

31. holy_father. "Antidetection", 2005. URL http://www.rootkit.com.

32. Holy_Father. "Hacker defender: readme", Nov 2005. URL http://www.hxdef.org.

33. Inc, Imperva. "Signature Detection", 2006. URL http://www.imperva.com/application_defense_center/glossary/signature_detection.html.

34. Inc, McAfee. "McAfee Avert Labs Points to Increasing Prevalence of Stealth Technology in Malware (Rootkits)", 2006. URL http://phx.corporate-ir.net/phoenix.zhtml?c=104920&p=irol-newsArticle&ID=843059&highlight.

35. Inc, McAfee. "Rootkits, Part 1 of 3: The Growing Threat", 2006. URL http://www.mcafee.com/us/local_content/white_papers/threat_center/wp_akapoor_rootkits1_en.pdf#search%22linux%20rootkits%20hiding%20techniques%22.

36. Intel. "Intel Architecture Software Developers Manual", 1997. URL http://developer.intel.com/design/pentium/manuals/24319001.pdf.

37. Jelena Mirkovic, Peter Reiher, Janice Martin. "A Taxonomy of DDoS Attacks and DDoS Defense Mechanisms", 2006. URL http://lasr.cs.ucla.edu/ddos/ucla_tech_report_020018.pdf.

38. Kay, Roger L. "How to Implement Trusted Computing", 2006. URL https://www.trustedcomputinggroup.org/news/Industry_Data/Implementing_Trusted_Computing_RK.pdf.

39. LINDSTROM, PETE. "A Patch in Time", Feb 2004. URL http://infosecuritymag.techtarget.com/ss/0,295796,sid6_iss326_art580,00.html.

40. Livingston, Brian. "IceSword Author Speaks Out On 'Rootkits"', Jun 2005. URL http://itmanagement.earthweb.com/columns/executive_tech/article.php/3512621.

41. Mar-Elia, Darren. "How the Registry Is Architected", 2000. URL http://www.windowsitlibrary.com/Content/224/3.html.

42. Martin, Michael J. "Router Expert: Understanding TCP/IP to prevent network attacks, part 2", 2002.

43. McKusick, Marshall Kirk and George V Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2005.

44. Mellon, Carnegie. "CERT/CC Overview Incident and Vulnerability Trends", 7 May 2003. URL http://www.cert.org/present/cert-overview-trends/module-2.pdf.

45. Merriam-Webster. "Steganography", 2007. URL http://webster.com/dictionary/steganography.

46. NSA. "Security-Enhanced Linux", 2007. URL http://www.nsa.gov/selinux/.

47. for Operating Systems, SIGOPS:Special Interest Group. "The i386 Interrupt Descriptor Table", 2002. URL http://www.acm.uiuc.edu/sigops/roll_your_own/i386/idt.html.

48. Picasso, Vilkatla, Layzer, Auriell, 99none, Phancy, and Krzysieq. "GMER", 2007. URL http://www.gmer.net/index.php.

49. Radcliff, Deborah. "Companies adapt to a zero day world", 13 Jul 2004. URL http://www.securityfocus.com/news/9100.

50. RobD. "Re:NDIS.sys", 2004. URL http://www.pcreview.co.uk/forums/thread-466431.php.

51. Russinovich, Mark. "DebugView", 2006. URL http://www.sysinternals.com/Utilities/DebugView.html.

52. Russinovich, Mark. "Process Explorer", 2006. URL http://www.sysinternals.com/Utilities/ProcessExplorer.html.

53. Russinovich, Mark E and David A Solomon. *Microsoft Windows Internals, Fourth Edition*. Microsoft Press, 2005.

54. Rutkowska, Joanna. *Introducing Stealth Malware Taxonomy*. Technical report, COSEINC Advanced Malware Labs, Nov 2006. URL http://www.net-security.org/dl/articles/malware-taxonomy.pdf.

55. Rutkowska, Joanna. "Subverting Vista Kernel For Fun And Profit", 2007. URL http://invisiblethings.org/papers/joanna%20rutkowska%20-%20subverting%20vista%20kernel.ppt.

56. Rutkowski, Jan Krzysztof. *Advanced Windows 2000 Rootkit Detection (Execution Path Analysis)*. Technical report, Jul 2003. URL http://www.blackhat.com/presentations/bh-usa-03/bh-us-03-rutkowski/bh-us-03-rutkowski-paper.pdf.

57. Schneier, Bruce. "Attack Trees", Dec 1999. URL http://www.schneier.com/paper-attacktrees-ddj-ft.html.

58. Schoen, Seth. "Trusted Computing: Promise and Risk", Oct 2003. URL http://www.eff.org/Infrastructure/trusted_computing/20031001_tc.php.

59. Silberschatz, Abraham, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons. INC, 2005. URL http://www.cs.pu.edu.tw/ ychu/class951/OperatingSystem/vendor/ch21.ppt.

60. Silberschatz, Abraham, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons. INC, 2005. URL http://www.cs.pu.edu.tw/ ychu/class951/OperatingSystem/vendor/ch22.ppt.

61. Soeder, Derek and Ryan Permeh. "eEye BootRoot", 2005. URL http://www.eeye.com/html/resources/downloads/other/index.html.

62. TheRealAphex. "AFX Rootkit 2005", 2007. URL http://www.rootkit.com/vault/therealaphex/AFXRootkit2005.zip.

63. Uninformed.org. "Windows OpenProcess", Jan 2006. URL http://uninformed.org/index.cgi?v=3&a=7&p=5.

64. Wichmann, Rainer. "Linux Kernel Rootkits", 2002. URL http://www.la-samhna.de/library/rootkits/basics.html.

65. Wikipedia. "Operating System", 26 Nov 2006. URL http://en.wikipedia.org/wiki/Operating_system.

66. Wikipedia. "Privilege (computer science)", 3 Sep 2006. URL http://en.wikipedia.org/wiki/Privilege_%28computer_science%29.

67. Wikipedia. "ntoskrnl.exe", 2007. URL http://www.answers.com/topic/ntoskrnl-exe.

68. www.trustedbsd.org. "TrustedBSD Project", 2007. URL http://www.trustedbsd.org/.

69. xshadow. "Vanquish v0.2.1: readme", 2005. URL http://www.rootkit.com.

70. Zovi, Dino Dai. "Kernel Rootkits", Jul 2001. URL http://www.theta44.org/lkr.pdf.

Glossary

Rootkit Tools designed to create and maintain an environment on a computer in which attack tools and activities may be hidden, such that a user does not know of their presence on a compromised machine.,

Signature based detection Data is scanned for a pattern that comprise a "fingerprint" that is unique to a particular entity [11].,

Heuristic/behavioral based detection Behavioral based detection seeks to identify actions or patterns that are abnormal to system operation. These detection techniques "work by recognizing deviations in "normal" system patterns or behaviors" [11].,

Crossview based detection Crossview based detection uses the idea of data redundancy (the existence of equivalent data in more than one location) to find "answers" from multiple sources that should be the same in order to identify discrepancies. An example of such is a child asking her mother for permission to go to the park and then asking her father permission for the same thing. The two answers should be the same; however, the difference in answers is what is exploited by the child. Detection techniques would use the difference in answers to identify the existence of a rootkit [11].,

Integrity detection Integrity based detection compares a known good entity with a suspected entity in order to verify the correctness/accurateness of the suspect entity. An example of such is comparing "a current snapshot of the filesystem or memory with a known, trusted baseline" [11].,

Hardware based detection

Hardware based detection uses a piece of hardware to implement signature, heuristic, crossview, or integrity based detection while separating itself from the suspected operating system [11].,

Steganography

Steganography as quoted from the Mirriam-Webster's online dictionary is "the art or practice of concealing a message, image, or file within another message, image, or file" [45] However, a short definition which describes how steganography works is: hiding in plain sight.,

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
22–03–2007	Master's Thesis	Sept 2005 — Mar 2007

4. TITLE AND SUBTITLE

A Study of Rootkit Stealth Techniques
and Associated Detection Methods

5a. CONTRACT NUMBER

DACA99–99–C–9999

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Daniel Nerenberg, 1Lt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, Bldg 640
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCE/ENG/07-10

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Dr. David Kapp
AT-SPI Technology Office AFRL/SN
2241 Avionics Circle
WPAFB, OH 45433-7320
(937) 320-9068
David.Kapp@wpafb.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approval for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In today's world of advanced computing power at the fingertips of any user, we must constantly think of computer security. Information is power and this power is had within our computer systems. If we can not trust the information within our computer systems then we can not properly wield the power that comes from such information. Rootkits are software programs that are designed to develop and maintain an environment in which malware may hide on a computer system after successful compromise of that computer system. Rootkits cut at the very foundation of the trust that we put in our information and subsequent power. This thesis seeks to understand rootkit hiding techniques, rootkit finding techniques and develops attack trees and defense trees in order to help us identify deficiencies in detection to further increase the trust in our information systems.

15. SUBJECT TERMS

rootkit, attack tree, defense tree

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Major Paul D. Williams
U	U	U	UU	145	19b. TELEPHONE NUMBER *(include area code)* (937) 255–3636, ext 7253 pwilliam@afit.edu